Commodore +4: Jim Butterfield's Memory Maps

CDC 00594

The Transactor

March 1985

The Tech/News Journal For Commodore Computers Vol. 5

90% Advertising Free!

Issue 05
$2.95

Hardware and Peripherals

- Revolutionary Memory Advancements
- The Commodore 64: An Inside Look
- Scanning The Keyboard Matrix
- Commodore 64 Keyboard Kernel Explained
- Keyboard Service and Repair
- Build Your Own Computer Desk
- Computer Slide Projector Control
- How DOS Works: Complete Dialect Details
- Choosing A Printer: Decision Criteria
- Evolution of The CPU
- EPROM Cartridge For The VIC 20
- TransBASIC: Custom Command Utility
- Linked Lists: Sort Without Sorting
- Plus Lots More, And ALL On Commodore!

03
0 72246 00594

INTRODUCING

THE PRO-LINE TEAM

PAL 64
The fastest and easiest to use assembler for the Commodore 64.† Pal 64 enables the user to perform assembly language programming using the standard MOS mnemonics. **$69.95**

POWER 64
Is an absolutely indispensible aid to the programmer using Commodore 64 BASIC. Power 64 turbo-charges resident BASIC with dozens of new super useful commands like MERGE, UNDO, TEST and DISK as well as all the old standbys such as RENUM and SEARCH & REPLACE. Includes MorePower 64. **$69.95**

TOOL BOX 64
Is the ultimate programmer's utility package. Includes Pal 64 assembler and Power 64 BASIC soup-up kit all together in one fully integrated and economical package. **$129.95**

SPELLPRO 64
Is an easy to use spelling checker with a standard dictionary expandable to 25,000 words. SpellPro 64 quickly adapts itself to your personal vocabulary and business jargon allowing you to add and delete words to/from the dictionary, edit documents to correct unrecognized words and output lists of unrecognized words to printer or screen. SpellPro 64 was designed to work with the WordPro Series* and other wordprocessing programs using the WordPro file format. **$69.95**

NOW SHIPPING!!!

For Your Nearest Dealer Call (416) 273-6350

WP64
This brand new offering from the originators of the WordPro Series* brings professional wordprocessing to the Commodore 64 for the first time. Two years under development, WP64 features 100% proportional printing capability as well as 40/80 column display, automatic word wrap, two column printing, alternate paging for headers & footers, four way scrolling, extra text area and a brand new 'OOPS' buffer that magically brings back text deleted in error. All you ever dreamed of in a wordprocessor program, WP64 sets a new high standard for the software industry to meet. **$69.95**

MAILPRO 64
A new generation of data organizer and list manager, MailPro 64 is the easiest of all to learn and use. Handles up to 4,000 records on one disk, prints multiple labels across, does minor text editing ie: setting up invoices. Best of all, MailPro 64 resides entirely within memory so you don't have to constantly juggle disks like you must with other data base managers for the Commodore 64. . **$69.95**

†Commodore 64 and Commodore are trademarks of Commodore Business Machines Inc.

*Presently marketed by Professional Software Inc.

Specifications subject to change without notice...

PRO·LINE SOFTWARE

(416) 273-6350
755 THE QUEENSWAY EAST, UNIT 8,
MISSISSAUGA, ONTARIO, CANADA, L4Y 4C5

Volume 5
Issue 05
Circulation 62,000

The Transactor

Transcriptions Editorial 3

News BRK . . . 4

The Diskette Transactor
The Complete Commodore Inner Space Anthology
Memorex Catalog
Helko Systems Furniture
Network Communications Show
EDG Electronics Announces Availability of Compuserve Software in Canada.
Frantek Software distributors for Scarborough Systems.
Commodore 64 Enhancements
Commodore to Distribute New Handic Software & Networking Devices
New Recreational Software for the +4 and C16
COMAL: Program For The Future – Today!
TPUG Offers COMAL Primer and Manual
EnTech Introduces Enhanced Studio 64
Melcher Software Announces Two Music Tutor Programs
Two Presidential Election Programs
RECORD BOOK: Data Storage and Retrieval
How To Make Good Investments
Investment and Statistical Software Package
MICROSHARE 64 networking system
Operant Interface
Electronic Components Computer Patch Cord
Omnitronix Announces RS232 Interface for the VIC–20, C64, and SX64 Portable.
RS–232 16 Channel Analog Input Module

Bits and Pieces 11

Built–in debugging aid
Easy Disk Directory Pattern matching
Poison Line Number
Closing "Forgotten" Files
SAVE–ing a Range of Memory From BASIC
WAIT A SECOND!
Checking for SHIFT, CTRL, and Commodore keys
Changing Screen Character Colours
Death by Garbage
Drowning in Garbage!
Single Disk Copy Program
BASIC 4.0 String Bug
Intercepting C64 System Error Messages
C64 RESTORE key checking
A Questionable Prompt
Fast BASIC HI–RES Point Plot
Fast HI–RES Screen Clear
Decimal to Hex conversion Table
Large Characters on VIC or 64

Letters 17

Doctor Destructo: Reset Protector
Coin Side One
Coin Side Two:

Transbloopers 19

The Commodore DOS: Two Book Reviews . 20
Machine Language For The Commodore 64 21
Commodore 16 / +4 Memory Maps 23
The MANAGER Column 26
Subroutine Eliminators 28
Introducing TransBASIC 30
Hardware Corner 36
Commodore 64 Keyboard Kernel Routines . 39
Fixing Commodore Keyboards 44
Life With The 1541 45
Learning The Language of DOS 46
Inside The Commodore 64 52
All About Printers 57
Evolution Of The CPU 62
EPROM Cartridge For The VIC 20 64
Computer Slide Projector Control 67
VIC 20 Audio/Video Cable Adapter 70
LINKED LISTS Part 1 72
Computing Desk 77
Rethinking DATAfication 78

The Transactor
The Tech/News Journal For Commodore Computers

Managing Editor
Karl J. H. Hildon

Editor
Richard Evers

Technical Editor
Chris Zamara

Art Director
John Mostacci

Administration & Subscriptions
Lana Humphries

Contributing Writers
Harold Anderson
Don Bell
Daniel Bingamon
Jim Butterfield
Gary Cobb
Elizabeth Deal
Domenic DeFrancesco
G. Denis
Brian Dobbs
Bob Drake
Ted Evers
Mike Forani
Jeff Goebel
Dave Gzik
Thomas Henry
David A. Hook
Phil Honsinger
Rick Illes
Scott Johnson
Garry Kiziak
Scott Maclean
Glen Pearce
Michael Quigley
Howard Rotenberg
Louis F. Sander
George Shirinian
K. Murray Smith
Darren J. Spruyt
Aubrey Stanley
Nick Sullivan
Colin Thompson
Mike Todd
James Whitewood

Production
Attic Typesetting Ltd.

Printing
Printed in Canada by
MacLean Hunter Printing

The Transactor is published bi-monthly by Transactor Publishing Inc., 500 Steeles Avenue, Milton, Ontario, L9T 3P7. Canadian Second Class mail registration number **6342.** USPS **725–050,** Second Class postage paid at Buffalo, NY, for U.S. subscribers. U.S. Postmasters: send address changes to The Transactor, 277 Linwood Avenue, Buffalo, NY, 14209, 716-884-0630. **ISSN# 0827-2530.**

The Transactor is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names (PET, CBM, VIC, 64) are registered trademarks of Commodore Inc.

Subscriptions:
Canada $15 Cdn. U.S.A. $15 US. All other $21 US.
Air Mail (Overseas only) $40 US. ($4.15 postage/issue)

Send all subscriptions to: The Transactor, Subscriptions Department, 500 Steeles Avenue, Milton, Ontario, Canada, L9T 3P7, 416 876 4741. From Toronto call 826 1662. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to Milton HQ.

Back Issues: $4.50 each. Order all back issues from Milton HQ.
SOLD OUT: The Best of The Transactor Volumes 1 & 2, and Volume 4, Issues 04 & 05
Still Available: Best of The Transactor Vol. 3, Vol. 4: 01, 02, 03, 06, Vol. 5: 01, 02, 03, 04

Program Listings In The Transactor

All programs listed in The Transactor will appear as they would on your screen in Upper/Lower case mode. To clarify two potential character mix-ups, zeroes will appear as '0' and the letter "o" will of course be in lower case. Secondly, the lower case L ('l') has a flat top as opposed to the number 1 which has an angled top.

Many programs will contain reverse video characters that represent cursor movements, colours, or function keys. These will also be shown exactly as they would appear on your screen, but they're listed here for reference. Also remember: CTRL-q within quotes is identical to a Cursor Down, et al.

Occasionally programs will contain lines that show consecutive spaces. Often the number of spaces you insert will not be critical to correct operation of the program. When it is, the required number of spaces will be shown. For example:

print" flush right" – would be shown as – print"[space10]flush right"

Cursor Characters For PET / CBM / VIC / 64

Down	– q	**Insert**	– T
Up	– Q	**Delete**	– t
Right	– [■]	**Clear Scrn**	– S
Left	– [Lft]	**Home**	– s
RVS	– r	**STOP**	– c
RVS Off	– R		

Colour Characters For VIC / 64

Black	– P	**Orange**	– A
White	– e	**Brown**	– U
Red	– £	**Lt. Red**	– V
Cyan	– [Cyn]	**Grey 1**	– W
Purple	– [Pur]	**Grey 2**	– X
Green	– ↑	**Lt. Green**	– Y
Blue	– ←	**Lt. Blue**	– Z
Yellow	– [Yel]	**Grey 3**	– [Gr3]

Function Keys For VIC / 64

F1	– E	**F5**	– G
F2	– I	**F6**	– K
F3	– F	**F7**	– H
F4	– J	**F8**	– L

Quantity Orders:

MICRON DISTRIBUTING

CompuLit
PO Box 352
Port Coquitlam, BC
V5C 4K6
604 438 8854

Micron Distributing
409 Queen Street West
Toronto, Ontario, M5V 2A5
(416) 593 9862
Dealer Inquiries ONLY:
1 800 268 9052
Subscription related inquiries
are handled ONLY at Milton HQ

U.S.A. Distributor:

Capital distributing

Capital Distributing
Charlton Building
Derby, CT
06418
(203) 735 3381
(or your local wholesaler)

Master Media
261 Wyecroft Road
Oakville, Ontario
L6J 5B4
(416) 842 1555
(or your local wholesaler)

Editorial contributions are always welcome. Writers are encouraged to prepare material according to themes as shown in Editorial Schedule (see list near the end of this issue). Remuneration is $40 per printed page. Preferred media is 1541, 2031, 4040, 8050, or 8250 diskettes with WordPro, WordCraft, Superscript, or SEQ text files. Program listings over 20 lines should be provided on disk or tape. Manuscripts should be typewritten, double spaced, with special characters or formats clearly marked. Photos of authors or equipment, and illustrations will be included with articles depending on quality. Diskettes, tapes and/or photos will be returned on request.

All material accepted becomes the property of The Transactor. All material is copyright by Transactor Publications Inc. Reproduction in any form without permission is in violation of applicable laws. Please re-confirm any permissions granted prior to this notice. Solicited material is accepted on an all rights basis only. Write to the Milton address for a writers package.
The opinions expressed in contributed articles are not necessarily those of The Transactor. Although accuracy is a major objective, The Transactor cannot assume liability for errors in articles or programs. Programs listed in The Transactor are public domain; free to copy, not to sell.

Transcriptions

We're So Misunderstood!

A couple of issues back we published a "mock ad" type cartoon describing the "Comedian 264". Since then we've heard reports ranging from tickled to mild shock. Well, we at The Transactor would like to set the record straight. Before anyone else is even the least bit influenced by what was merely intended to be a little 'Don Rickles' style humor, here's how we see the picture to this point in the life of the new Commodore +4.

To start, it's only natural that Commodore will dub the newest machine their "latest and greatest". They have to do that. It's called, "staying in business". But now it seems that many Commodore 64 owners are showing what you might call "technology territorial" behaviour patterns. The 64 has features not found in the +4. So to maintain a technologically superior feeling towards their investment, some 64 owners will naturally see a new machines weaknesses over its strengths. Besides, the "new kid on the block" is bound to take a little ribbing, but that stage is pretty well over.

The 64 and the +4 have many similarities. But the advanced features unique to each machine will have the most influence over a decision between the two. That is, if money is no object. The +4 will cost more than the 64 which is more than the C16, the 16K version of the +4 in a C64 casing.

The 64 will remain the superior entertainment machine. The animated graphics capabilities will ensure that. The +4 has a SOUND command, but music applications of any sophisticated nature will still be superior with the SID chip. And Commodore is not about to give their star player its walking papers.

The +4 wins in the "productivity" department, the main promotional base for the machine. However, if that was your main reason for owning a 64, remember, a good investment is one that does the job it was intended to. There will always be another new computer, but you can't wait forever. If you know you need one, it's a question of how long you can go without. You must determine how valuable a computer will be for the task at hand, then subtract that value over the period of time you spend waiting. Many Commodore 64 owners using "productivity" software will probably agree that waiting would have been a mistake. In fact, it's a good possibility that those machines have not only paid for themselves, but also a brand new +4 system.

The 16 has been dubbed "The Learning Machine" and will no doubt be a serious contender in the battle for the educational market. School budgets should have little trouble with 99 bucks U.S.

VIC 20 prices are pretty well rock–bottom with reports as low as $49 in Florida. Apparently Commodore still has inventory, but the VIC 20 stamp has probably seen the production line for the last time.

That's our account of the situation. Now the machine itself.

Quite simply, the Commodore 16 and +4 generation of computers are nothing less than fabulous! In fact, from the aspect of programmability, it's the best Commodore machine yet. And, as usual, dollar for dollar the other manufacturers don't even come close.

Sure it doesn't have a SID chip, but the SOUND command provides enough audio for the average program. It doesn't have sprites, but you don't use sprites to display business reports and analytical data.

New features like luminance control and flashing attributes are included; the MLM now has Assemble, Disassemble, the invaluable Hunt command, and more; Decimal and Hex converter programs are no longer necessary; the Editor is the best anywhere with features like Renumber, Delete, Help, Trace, Find, and Change; the new BASIC has all the structured commands you'll need; and with 60K of RAM available for BASIC means there will be no excuse for software that is anything less than exquisite!

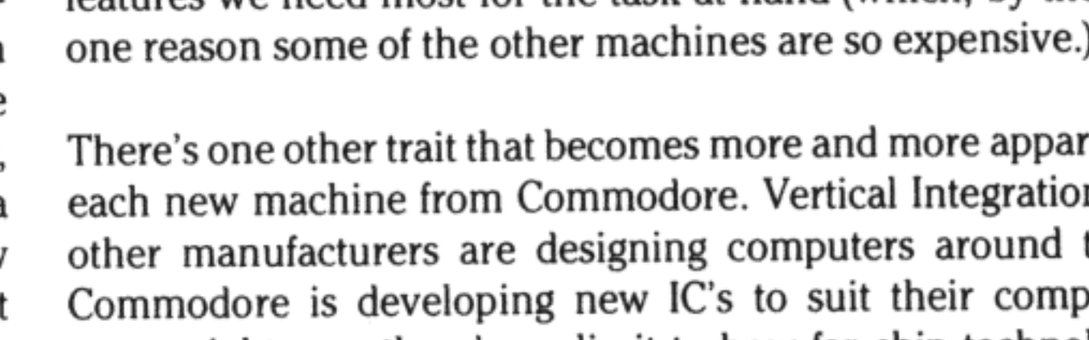
Even the new microprocessor technology of the 7501 makes it the leader in sophisticated architecture advancements. Multiple memory "layers" (see the Memory Map in this issue) allow for piles of ROM that effectively requires no address space - a concept that's not only clever, but truly intelligent!

The +4 might be the latest, but that doesn't make the 64 obsolete. For the application minded, either machine will be the right choice depending on the needs of the buyer, which is how a computer purchase should be made. For the hobbyist, personal preference will probably be the deciding factor. The introduction of the +4 fills a gap; we'll have the choice of an entertainment machine with business potential, or an applications machine with entertainment potential. In short, we now have the choice of paying only for the features we need most for the task at hand (which, by the way, is one reason some of the other machines are so expensive.)

There's one other trait that becomes more and more apparent with each new machine from Commodore. Vertical Integration. While other manufacturers are designing computers around the IC's, Commodore is developing new IC's to suit their computers. It seems right now there's no limit to how far chip technology can advance. As the IC gains more power, there will be less and less supporting hardware required on the same PC board which can only result in better, cheaper, and more reliable equipment. Commodore is well aware of their attributes and without a doubt they'll take full advantage of them in the years ahead.

Delivery of the new machines has just begun so it shouldn't be long before you see more and more material becoming available, and The Transactor will be no exception. Actually, we can hardly wait to get started! Now, if we could only get one. . .

There's nothing as constant as change, I remain,



Karl J.H. Hildon, Managing Editor

News BRK

Transactor News

The Diskette Transactor

Starting with this issue, the programs published in each Transactor will be available on disk! Recent price reductions of floppy disks and the advent of quality high-speed disk duplicating outfits have made possible a service that was once unfeasible, impractical, and too expensive.

Each disk release will contain a standard set of utility programs followed by the programs in the corresponding issue, plus any programs that may be useful or relevant to the theme of that issue, or perhaps some light entertainment. The diskettes also make it possible for us to present other potentially invaluable programs that might never have been obtained because they were too long to enter by hand.

No copylock protection will be added to the diskettes and faulty copies will be replaced free of charge.

Authors take note – depending on the success of this service, a royalty will be paid each time your program is responsible for the order.

The Price? Just $9.95 Canadian for your first disk! With every disk shipped, a personalized "money off" coupon will be included good for $2 OFF the purchase of any Diskette Transactor – that's just $7.95 CDN or $5.95 U.S! Return the coupon and we'll send you the disk for any issue you choose (>=Vol 5, Issue 5) and the same coupon for another $2 discount (until it gets worn out of course). If you decide you don't want the diskette for a certain issue, keep the coupon until the next issue. This way you have the choice of ordering them issue-by-issue, only at *subscription* prices.

The first Diskette Transactor will coincide with Volume 5, Issue 05, (this issue) and will be available by December 15. All future diskettes will be ready the same day the magazine is released (see Editorial Schedule). To order, send $9.95 Cdn (Ontario residents please add 7% sales tax) or $7.50 U.S. or your Visa/Master Card number (include Expiry Date) to:

The Diskette Transactor
500 Steeles Avenue
Milton, Ontario
L9T 3P7

Diskettes are 4040/1541 or compatible (ie. MSD) format. With enough response we'll offer 8050 compatible disks too.

We also plan to pack as many programs as we can from past Transactors onto a set of 2 or more diskettes. But until we know how many disks will be filled up, a price cannot be determined. See News BRK next issue for details.

The Complete Commodore Inner Space Anthology

That's the name we've settled on for the much awaited second edition of Transactor Reference Issue. Inquiries for the book have been coming in from around the world, with some reports of people leaving full deposits with retailers.

At this point, the "Transcyclopedia" (as it's affectionately known) is well over 100 pages of strictly reference material. Whole subjects of information have been summarized and presented in a tidy, compact format that's been designed with the computer enthusiast in mind. Several pages contain as much as 20,000 characters each!

Sections include Music, Color and Graphics, Math & Conversion Formulae, High Level and Machine Language summaries, Hardware and Electronics, User Groups, Communications, Memory Maps for all machines including the B Series, +4/16, COMAL, and the 8050, 4040, and 1541 disk drives, Control Command summaries for all the most popular software, plus a mountain of general type info that usually can't be found all in one place. With any luck, we may also get permission to reprint VIC 20 and Commodore 64 schematics.

The price should be available by the time you read this or December 1, '84, whichever comes first. That's about the same date we expect to have the first copy off the press.

General News

Memorex Catalog

Memorex Canada, a complete supplier to the computer and word processing industry specializing in peripherals and accessories, is pleased to announce immediate availability of their new Computer Media and Communications catalogue.

In response to an increasing demand from their customer base, Memorex has produced a 24 page catalogue outlining the complete range of products available from Memorex. These include computer tape, disc packs, flexible discs, screens, controllers, and printers, as well as, a large selection of storage and ergonomic accessories for both computer and word processing environments.

Products may now be ordered via toll free phone lines across Canada at 1–800–268–9886.

Helko Systems Furniture

Helko is proud to announce the design and development of it's newest ergonomic workstation – the Micro Desk.

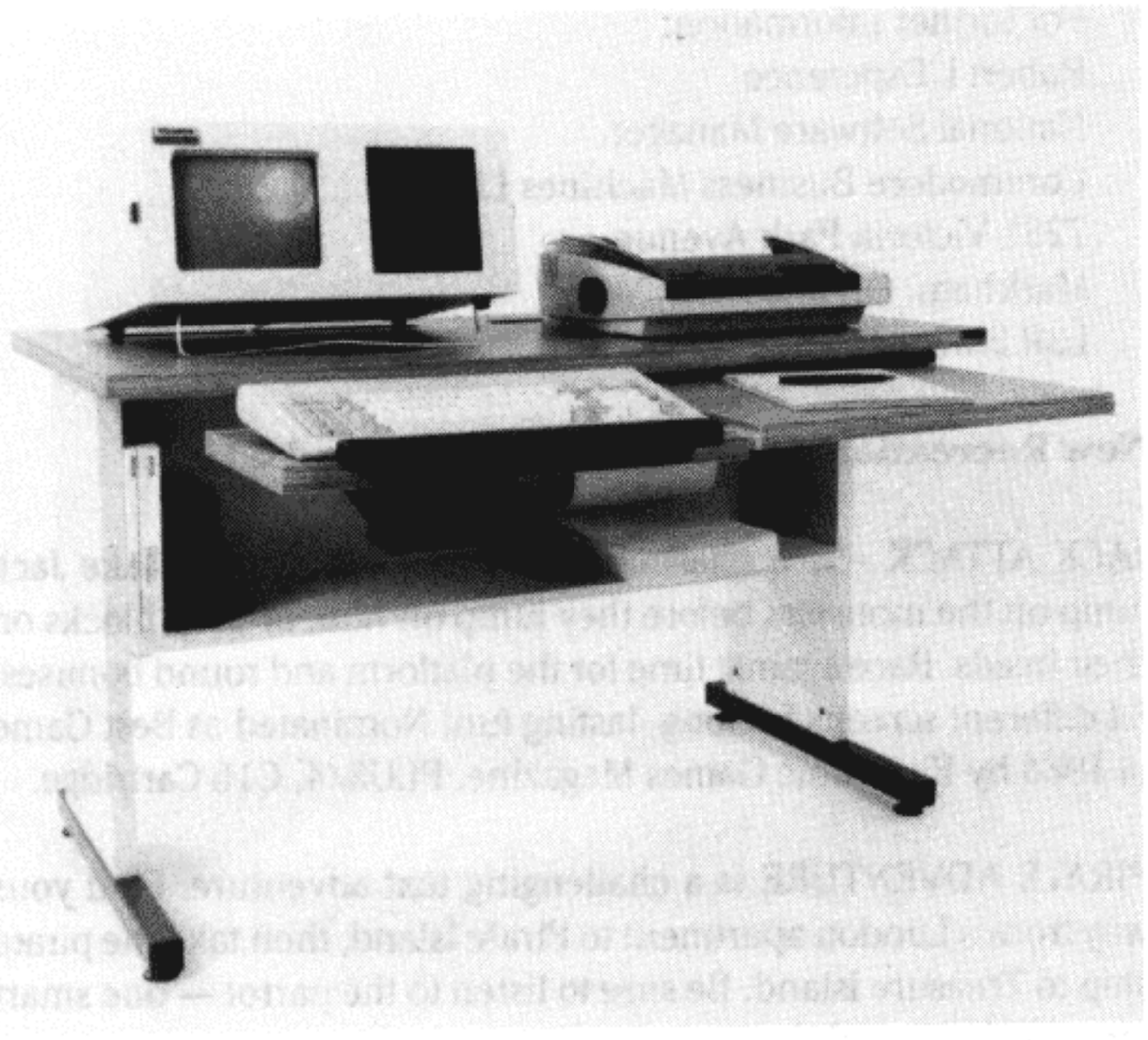
The Micro Desk provides the operator with a complete terminal support system and adequate working surface. The keyboard is fully adjustable in terms of height, tilt, depth and swivel. All variations to the keyboard location are easily accomplished from a seated position.

A pull out shelf provides the operator with additional workspace. It slides back under the unit when it is not required. A utility shelf underneath the unit and provides easily accessible storage for diskettes, operator manuals, paper etc.

The surfaces are constructed with 'melamine' – finished, three layer, two density, 45lb. PSI resistant particle board. All surfaces are wear, scratch and impact resistant. Feet are 16 gauge steel painted with brown two component apoxy for durability and a smooth finish. Adjustable glides permit fine levelling to compensate for uneven floors and minimize vibration.

Colors available are light oak, walnut and putty. For further information, please contact:

Gillian Oxley, Public Relations Director
Helko Systems Furniture Inc.
PO Box 712
Cornwall, Ontario
K6H 5T5 613 938–0494.



Events

Network Communications Show

The Communication Networks Conference & Exposition is being held January 28–31, 1985 at the Washington Convention Ctr., Washington, DC. Expo space rate will be $19.50/sq. ft.

The trade show will be featuring voice and data telecommunications, electronic mail, data processing, data communications and networking with a personal computer. For further information, contact:

Louise Myerow, Registration Manager
CW/Conference Management Group
375 Cochituate Road, Box 880
Framingham, MA 01701
617 879–0700 or 800 225–4698

Software News

EDG Electronics Announces Availability of Compuserve Software in Canada.

EDG Electronics is pleased to announce that it is now exclusive distributor in Canada for the products of Compuserve Information Services, of Columbus Ohio.

Compuserve is an easy–to–use videotex service designed for personal computer users and managed by the communications professionals who provide business information to over one fourth of the FORTUNE 500 companies.

Subscribers to the Consumer Information Service receive a wealth of useful, profitable and interesting information like national news wires, electronic banking, and shop at home services, and sophisticated financial data. There is also a communications network for electronic mail, a bulletin board for selling, swapping, and personal notices, and a multi–channel CB simulator.

Subscribers to the Executive Information Service receive serious, up–to–the minute business information including high–lighting investment information, communications, news and travel. The Executive Information Service also provides access to the Consumer Information Service. It provides InfoPlex electronic mail system, Ticker Retrieval for easy access of all information available on any company specified, true 20 minute delayed stock quotations, MicroQuote II, detailed demographic data, Executive News Service, plus more services to be announced.

Compuserve Vidtex Software provides terminal emulation for popular personal computers including Apple and Commodore. Compuserve with Vidtex offers options like colour graphics, automatic logon, full printer support, RAM buffer, user–defined function keys, cursor positioning and more.

Subscribers to either the Consumer Information Service or the Executive Information Service receive a subscription to Online Today, Compuserve's monthly magazine on videotext and information systems.

EDG Electronic Distributors is making Compuserve products available through a nation–wide network of computer stores. Potential

dealers, distributors and end users are invited to contact Ian Manson, Sales Manager, EDG Electronics for more information.

Electronic Distributors Inc.
3950 Chesswood Drive
Downsview, Ontario
M3J 2W6 416 636 9404

Frantek Software distributors for Scarborough Systems.

Frantek Software Distributors, Inc.; 1685 Russell Road; Ottawa; Ontario, has been named a distributor of home software in the educational and productivity areas by Scarborough Systems, Inc., Tarrytown, N.Y.

Frantek Software will distribute Scarborough's entire product line, including MASTERTYPE, the best-selling educational software program of all time; PHI BETA FILER, a list management program for children; RUN FOR THE MONEY, a business game with a space-age theme; YOUR PERSONAL NET WORTH, a home financial management program; SONGWRITER and PATTERNMAKER.

Please contact:
Sanford Bain
Scarborough Systems, Inc.
914 332-4545

Commodore News

Commodore 64 Enhancements

Late reports have it that a C128 is lined up in the Commodore queue. The C128 is, alledgedly, a Commodore 64 with 128K of RAM with the option of an 80 column black and white display. That's everything for now though - watch News BRK for details on price, availability, plus any other features that haven't been mentioned yet.

Commodore to Distribute New Handic Software & Networking Devices

Commodore Business Machines Limited, has announced a distribution agreement with Handic Software of Pennsuken, N.J. whereby Commodore will be exclusive distributor for Handic products in Canada.

Several new Handic cartridge software packages are now available for the Commodore 64.

Calc Result is a three-dimensional spreadsheet that can translate numbers into charts at the press of a button, and Word Result is a full-featured word processor that integrates with Calc Result.

Forth 64 is a powerful operating system with a programming language that is suitable for nearly every imaginable application in business as well as in process control environments. Stat 64 simplifies work with statistics and graphic displays. It adds 19 commands to the BASIC language, such as horizontal or vertical bar charts, plotting with 3871 points, screen dump and statistical commands for calculations of mean value, standard deviation, variance and more.

Graf 64 turns solutions of equations into graphical analysis. Users can define functions and plot the graph in high resolution with an X-axis range, and there is a special routine for computing the integral of a function within a range specified by the user.

Tool 64 is a powerful programming and debugging aid which includes numerous new commands, making it possible to write bigger and more advanced programs using much less code. Tool 64 has advanced input routines and excellent possibilities to handle numbers and to make graphics such as bar- and pie-charts.

The CBM-64 Relay Cartridge will allow C-64 owners to control burglar alarms, garage doors, door locks, electric radiators, lamps, transmitters, remote controls, valves, pumps, telephones, accumulators, irrigation systems, electric tools, stop watches, ventilators, air conditioners, humidifiers, miniature railroads and many other devices with their Commodore 64s.

Bridge 64 is a bridge game with numerous levels, from novice to advanced. Bridge 64 offers a helping partner or a skilled opponent with thousands of different bids and excellent graphics.

Commodore is also now distributing low-cost networking devices for its C-64, VIC-20 and PET/CBM models.

VIC-SWITCH allows up to eight Commodore 64s or VIC-20s to sequentially share one disk drive or printer. VIC-SWITCH was developed especially for educational use but has proven successful in every application where more than one VIC-20 OR C64 needs access to the same disk or the same printer.

PET-SWITCH allows up to 15 Commodore PET 4000 and/or CBM 8000 series micros to be connected to the same disk drive and printer. In a PET-SWITCH system, there is distributed data power, where data from the central disk drive, works with it and then saves the resulting data back to the disk.

PET-SWITCH is very useful in schools, offices, service bureaus and other applications where several people need continuous access to particular data. With PET-SWITCH this is achieved, while system costs can be cut considerably.

For further information:
Robert L'Esperance
National Software Manager
Commodore Business Machines Limited
7261 Victoria Park Avenue
Markham, Ontario.
L3R 2M7 416 475-5111

New Recreational Software for the +4 and C16

JACK ATTACK - is a charming video strategy game. Make Jack jump on the monsters before they jump on him, or drop blocks on their heads. Race against time for the platform and round bonuses. 64 different screens for long-lasting fun! Nominated as Best Game of 1984 by Electronic Games Magazine. PLUS/4, C16 Cartridge.

PIRATE ADVENTURE is a challenging text adventure. Find your way from a London apartment to Pirate Island, then take the pirate ship to Treasure Island. Be sure to listen to the parrot — one smart bird! PLUS/4, C16 Cartridge.

ATOMIC MISSION is a devious text adventure. You are locked inside a nuclear reactor which has been programmed by a mad scientist to explode. Your mission is to fool the security system and deactivate the bomb! PLUS/4, C16 Cartridge.

STRANGE ODYSSEY is a science fiction text adventure. Discover the secret to the machines in a space ship, and find the treasures hidden on the different worlds! PLUS/4, C16 Cartridge.

SCRIPT/PLUS is an advanced, full–featured, professional word processor complete with on-line help screens. It's features include wordwrap, full screen editing, global and local search and replace, automatic centering, justification and right alignment, and decimal alignment. There are automatic mail merge with selective search criteria, and file linking for easy document handling. One of the more advanced features is the capability for row and column arithmetic and column move. to simulate a spreadsheet or calculator within your document. SCRIPT/PLUS also contains an informative status line to help make your word processor easy to use. PLUS/4, C16 Cartridge.

FINANCIAL ADVISOR is a financial program designed for the high school/college finance student for the loan officer at the bank. FINANCIAL ADVISOR calculates the high cost and benefits of five common financial strategies: Periodic Deposit Accounts, Periodic Withdrawal Accounts, Installment Loans, Stocks and Bonds. Each financial strategy is broken down into pieces so you can change every facet of the loan or investment calculation. Both compounding and transaction periods can be changed quickly and easily. In addition, a special calculator mode lets you interrupt a problem to perform calculations, without disturbing your current work. There are 100 "memories" in which intermediate results may be saved. All calculations may be printed. PLUS/4 Cartridge.

LOGO is the computer language for everyone. It's simple enough for preschoolers and sophisticated enough to challenge college students. Special features of Logo includes Turtle Graphics which allows the user to draw pictures, Splitscreen which allows the user to watch their picture form as they are developing their program, Fullscreen which allows the user to see only their picture, and Edit for easy program corrections. The programmer can use either simple Logo commands or more complex procedure to create programs. Logo also uses the high and low resolution graphics, the 128 colors, and music capabilities of the Commodore PLUS/4 computer. The step by step manual makes learning Logo easy. PLUS/4 Cartridge and utility disk.

PLUS/4 TUTORIAL and C16 TUTORIAL is available as an introduction the the machines. In a friendly format, it teaches the keyboard and lets the user practice using the various keys. The tutorial has a step by step approach: various sections cover cursor control keys, alphabetic keys, graphic keys, colors and reverse keys, control keys such as escape and function keys, etc. The capabilities such as graphics, color, sound, flashing and window are used to make the tutor interesting and educationally effective. PLUS/4, C16 Cartridge.

COMAL: Program For The Future – Today!

Your computer is only as good as the programs it runs. Unleash the power built into your Commodore 64. Take control with COMAL, one of the fastest growing programming languages in the world.

COMAL is expertly designed and shines after a full 10 years of refinement. With COMAL you get only the best. The best of PASCAL; structures. The best of LOGO; turtle graphics. The best of MODULA 2; modules. The best of ADA; packages. And even with all the power and flexibility, it retains the best part of BASIC; ease of use. But best of all, it's fast and affordable.

While professional programmers marvel over the COMAL 2.0 Cartridge, most Commodore 64 users can benefit from COMAL 0.14 which already is part of most Commodore User Group Disk libraries. COMAL is here to stay as evidenced by the multitudes of programs and books available. There already are over 15 disks full of ready to run COMAL programs and 6 different COMAL books. By Christmas there should be another 10 disks of programs and 7 more books. There even is a newsletter, COMAL TODAY, devoted solely to COMAL, already 56 pages in its fourth issue.

However, the best news of all is that the COMAL 2.0 cartridge is already into mass production by Commodore in Europe and that the COMAL Users Group will be distributing it in the United States. This cartridge is the same size as a Commodore 8K game cartridge, yet inside is a complete 64k COMAL system in ROM. There is even an empty socket for an optional 8K, 16K, or 32K EPROM. The COMAL cartridge contains everything from the disk loaded COMAL 0.14 plus much more. In additional to all the fun of COMAL, the cartridge provides a professional language, with TRACE, ERROR HANDLERS, EXTERNAL PROCEDURES, BATCH COMMAND FILES, and 3 different screen dumps – all built in. No wonder Commodore can't keep up with the demand for the COMAL 2.0 Cartridge in Europe.

So now the choice is not whether to use COMAL or not, but which version? Get started with disk loaded COMAL 0.14 with the $19.95 Enhanced COMAL Pak, including 2 disks and a reference card, or the $29.95 COMAL Starter Kit which adds a book and third disk inside a nice case. But if only the best is good enough for you, then you will want the $99.95 COMAL 2.0 Cartridge package, including 2 books, 2 disks, and the cartridge in a custom molded case. If you can't make up your mind, send a Self Addressed Stamped Envelope for a FREE COMAL INFO PAK.

VISA and MasterCard orders can call toll–free 1–800 356–5324 extension 1307. Everything is available from the official source:

COMAL Users Group, U.S.A., Limited
5501 Groveland Terrace
Madison, WI
53716 608 222–4432

TPUG Offers COMAL Primer and Manual

TPUG (the world's largest Commodore users group) is pleased to announce the publication of a new COMAL programming manual, written by one of the language's creators, Borge Christensen.

COMAL is a high–level language with all the elements that make BASIC simple for the beginner but with additional features that make it easy to write well–structured programs.

The book contains over fifty pages of text and examples. It is ideally suited as a language tutorial as well as a desk–top reference manual for programmers. The cost is $9.95. For information on

this and more TPUG software, write to:

TPUG Inc.
1912A Avenue Road,
Suite #1
Toronto, Ontario. M5M 4A1

EnTech Introduces Enhanced Studio 64

Since its introduction in May, 1983, EnTech Software's Studio 64 has received critical acclaim and blossoming sales. Now, EnTech has given its most popular program for the Commodore 64 a major redesigning which will make it even easier and more flexible to use.

Studio 64 creates music as it is played on the Commodore 64 keyboard, and it is the only program that allows each note to have a different waveform sound. It can create songs up to eight minutes long which can be saved to disk or added to another program. Studio 64 now features notes in high resolution graphics and truer musical notation, including tied notes, sharps and flats. Control key functions make it easier to change voices and clefs, play back music and adjust sounds and filters. The new version also has twelve sample songs including "Billie Jean" and "Sweet Dreams".

With its many improvements, Studio 64 is still compatible with Add Mus'In, which adds music to any other program. The new Studio 64 will also play music created with the older version, and music created with either version will be accepted for EnTech's First Annual Computer Song Writing Contest.

Studio 64's major revisions were made by Ray Soular, a musician and record producer as well as EnTech's chairman. "This version was designed by the public rather than programmers," Soular said. "We've taken into consideration every possible suggestion to create the ultimate music synthesizer."

The new Studio 64 will retail at the same price as the original, $39.95. While the new version is available only on disk, the original version is still available on cassette. Current Studio 64 owners can upgrade to the new version by sending $10 and their old disk to:

EnTech Software
PO Box 185
Sun Valley, CA
91353 818 768-6646.

Melcher Software Announces Two Music Tutor Programs

The PIANO TUTOR and the VIOLIN TUTOR both enhance note reading ability and sound identification. The computer depicts note placement on the scale (treble and bass clef for the piano), sounds the note, and asks for the note identification. The student's score is computed for speed and accuracy. Colorful graphics and stimulating sound make these programs dynamic tools for today's ambitious music student.

The cost of the above programs is $29.95 for RECIPES SUPREME and $19.95 for either the PIANO TUTOR or the VIOLIN TUTOR. If purchased together, both music programs are available for $34.95. Add $3.00 to total order for postage and handling. MELCHER SOFTWARE welcome inquiries about these programs. Contact:

Melvin Billik
MELCHER SOFTWARE
PO Box 213
Midland, MI
48640 517 631-7607.

MicroEd Offers Two Presidential Election Programs

MicroEd Incorporated, publishers of more than one thousand educational programs, is currently making a special marketing effort on behalf of two presidential election programs for the Commodore 64: HAT IN THE RING and THE AMERICAN PRESIDENCY.

Hat in the Ring is a two-player (or two-team) exercise designed to acquaint students with some of the political considerations involved in running a presidential candidate – one for the Republicans, the other for the Democrats. Throughout the exercise, each candidate works the the factors of media exposure, personal campaigning, domestic issues, and international issues in order to make decisions that will result in a successful campaign. Suggested retail price for the program is $9.95.

The American Presidency is a set of four programs designed to present general information about, and a historical review of, the office and those who have held it. Included is a treatment of THE OFFICE, THE DUTIES, THE PRESIDENT 81788-1888), and THE PRESIDENTS 81892-1980). A one- or two-player (or team) game is attached to each of the instructional lessons. The suggested retail price for the 4-program set is #34.95.

Persons desiring further information may call George Esbensen at toll free 1-800-MicroEd.

MicroEd, Inc.
PO Box 444005
Eden Prairie, Minnesota
55344 612 944 8750

RECORD BOOK: Data Storage and Retrieval System for ALL Commodore computers

Many computer owners ask "What can my computer do for me? Record Book provides an answer.

Record Book allows you to store any categorized information on disk. It is ideal for keeping phone numbers, mailing lists, home inventory, and even some business records (like outstanding debts, supplier lists, inventory). Once information is put into a file, it can be searched through, changed, added to, deleted, or sorted. All of these functions are at the touch of a few keys. Record Book also supports printing of all or selected parts of a file to any properly interfaced printer.

Record Book requires only 16K RAM, 40 (or 80) columns and, of course, a disk drive. A VIC-20 can fill this bill, if it is expanded with 11K RAM and a 40-column adapter.

Record Book comes with a detailed manual which describes all Record Book functions and even takes you on a 'live guided tour' through some of the most–used features. Record Book is completely menu–driven, which means that the novice user cannot get lost or forget a command, but demanding users will find its many functions advanced and convenient. If you have data, record Book will handle it.

Record Book costs $25.00, but it does what far more expensive databases do.

The author runs a bulletin board (accessible only by 300 baud modem) at 416 239–5993. He can be reached there as well as at the above address.

Madill and Welsh Computing
7 Strath Humber Court
Islington, Ontario M9A 4C8

How To Make Good Investments

Computer aided instruction for the Commodore 64, geared to teach you the fundamentals of stock market and real estate investment analysis, is the first course in a series of courses on investment and financial analysis developed by experienced professionals from the top business schools.

The courses are designed to cover the same material as is covered in the best business schools with some practical street techniques. The investment techniques in the initial course were selected for their ease of use and understanding. Programs and examples using those programs are provided as learning aids and for subsequent investment analysis.

Course I: HOW TO MAKE GOOD INVESTMENTS" comes complete with a 100 page text and programs on disk for $54.95. This course and additional courses may be tax deductible. (Commodore 64 is a registered trademark of Commodore Electronics Ltd.) Send $54.95 check or money order to:

Course I
The Wizards
PO Box 7118
The Woodlands, TX
77387

Investment and Statistical Software Package

Commodore Users can now gain access to the leading investment and statistical software used by IBM and other computers.

Programmed Press announces that its Investment and Statistical Software Package, containing 50 programs for Statistical Forecasting, Stocks, Bonds, Options, Futures and Foreign Exchange, is "ready–to–run" on the C–64.

The 220 page Computer–Assisted Investment Handbook by Dr. Albert Bookbinder lists, explains and gives sample RUN illustrations for all fifty BASIC programs for profitable planning and forecasting.

This software supports Commodore, Apple, Radio Shack, and IBM personal computers, as well as other machines using MS–DOS.

Commodore 64 requires only one disk drive. The price is $100 for the "ready–to–run" Investment and Statistical Software and only $19.95 for the Handbook that lists all 50 programs. VISA and MasterCards are accepted.

Programmed Press
2301 Baylis Ave.,
Elmont, NY
11003 516 775–0933

Hardware News

MICROSHARE 64 networking system

The Microshare 64 is a networking system that allows up to 8 C64s to be connected to a common IEEE or serial bus. The system can be configured for 8 computers and an IEEE bus, or 7 computers with both IEEE and serial. Users are given access to the bus on a "first come, first served" queuing basis, the next user being selected when the bus remains idle for a chosen period of time. A 14K print buffer is built in to free up the network when a user sends output to the printer. A unique capability of the Microshare 64 is it's "group load" feature (a.k.a. "megaload"), which allows any or all of the computers in the network to simultaneously LOAD the same program from disk. Some other features include: Individual disk error status reports for each user, individually controllable channel switching delay, built in diagnostic routines, and sturdy all–steel construction.

The price for the basic unit (computer–connection cables included) is $995.00 (Canadian + FST). Available from:

Comspec Communications Inc.
153 Bridgeland Ave., Unit 5
Toronto, Ontario
M6A 2Y6 416 787–0617

Operant Interface

Psychronix has released the Model 35 Interface for Psychologists who want to record up to 3 separate switch closures, and turn on 4 lamps and a feeder in a single 2BVDC operant chamber.

This interface uses the parallel port on the VIC–20, C–64, or PET/CBM computer. Because it is programmable entirely in BASIC (machine language programming is not necessary) students can be quickly taught to use this interface. Data can be recorded in real time using the computer's internal clock.

The Model 35 Interface has been in use for over a year in a research laboratory with remarkable success. All chips are socketed for easy replacement. Retail price of the interface is $130.00

Psychronix
Box 422
Logan, UT 84321

Electronic Components Announces New Computer Patch Cord (CPC–1000)

Just developed and patented, a new product allows the owner of a Commodore 64 or VIC–20 personal computer, to use an ordinary cassette recorder with their computer. The Computer Patch Cord (CPC–1000) is compatible with everything from the Emerson 4940 Ghetto Blaster to the Sony Walkman 11.

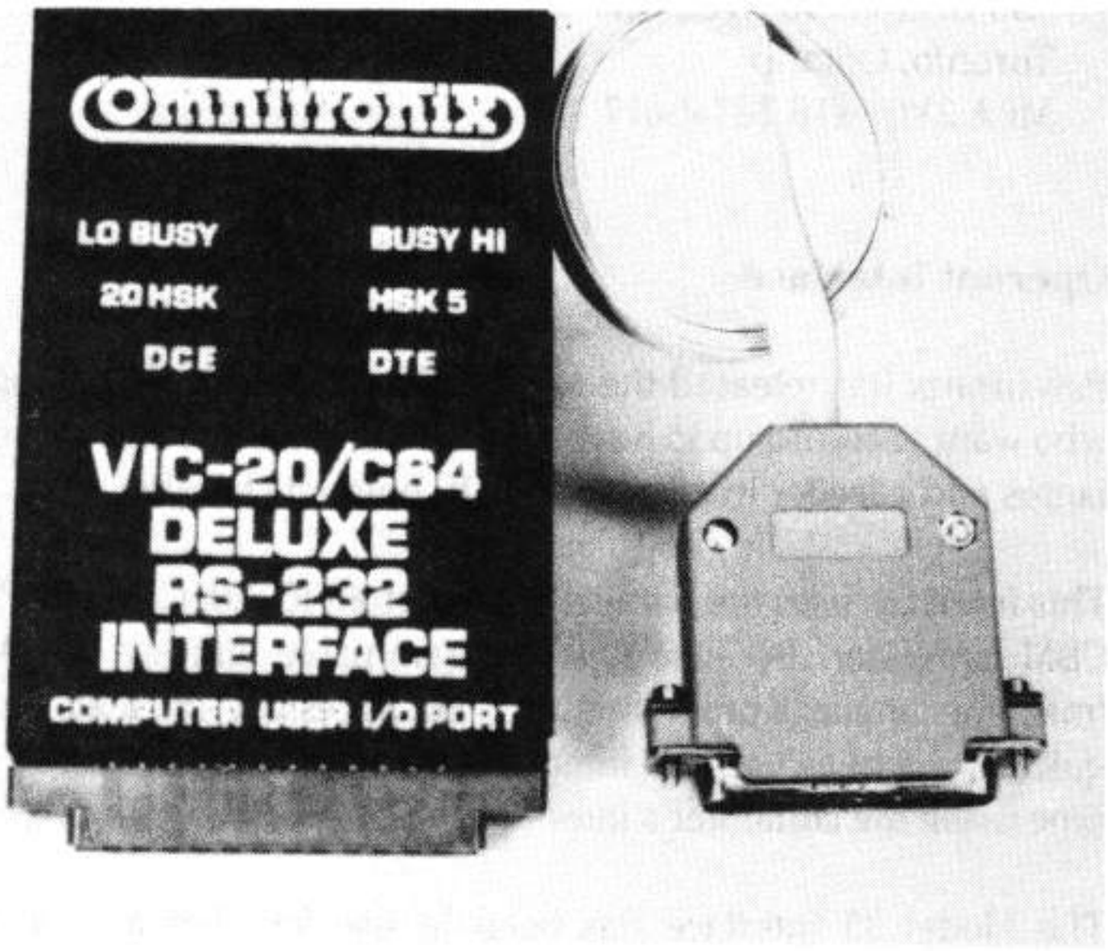
The advantage of the CPC–1000 is that it costs a fraction of what a Commodore Datasette would cost. The average price of the Commodore Datasette is $79.95, the price of the Computer Patch Cord is only $29.95. For further information please contact:

Brian G. Wilcox, Vice President
Electronic Components
P.O. Box 173
Elma, N.Y. 14059

Omnitronix Announces RS232 Interface for the VIC–20, C64, and SX64 Portable.

The RS232 Interface has been specially designed to allow easy use of any type of RS232 equipment, including serial printers and modems. The Deluxe RS232 Interface plugs into the USER I/O port of the computer. Included as part of the unit is a three foot cable terminating in a male or female DB25 connector. The Deluxe RS232 Interface can also be supplied with a PC Board mounting Female DB25, allowing it to completely substitute for the 1011A. Three switches in the case cover allow you to set the unit for DTE/DCE, invert pins 20 and 5, and select the BUSY line polarity.

The RS232 Interface supports virtually all RS232 signals including Ring Detect. It can operate at up to 2400 baud. Supplied in the manual is a Type–In BASIC terminal program and a very complete tutorial on using the RS232 port.



The unit has an unconditional 1 year guarantee, and a 30 day return if not satisfied. Suggested retail is $39.95. It is available from your local dealers or call Omintronix. Please add $1.60 shipping.

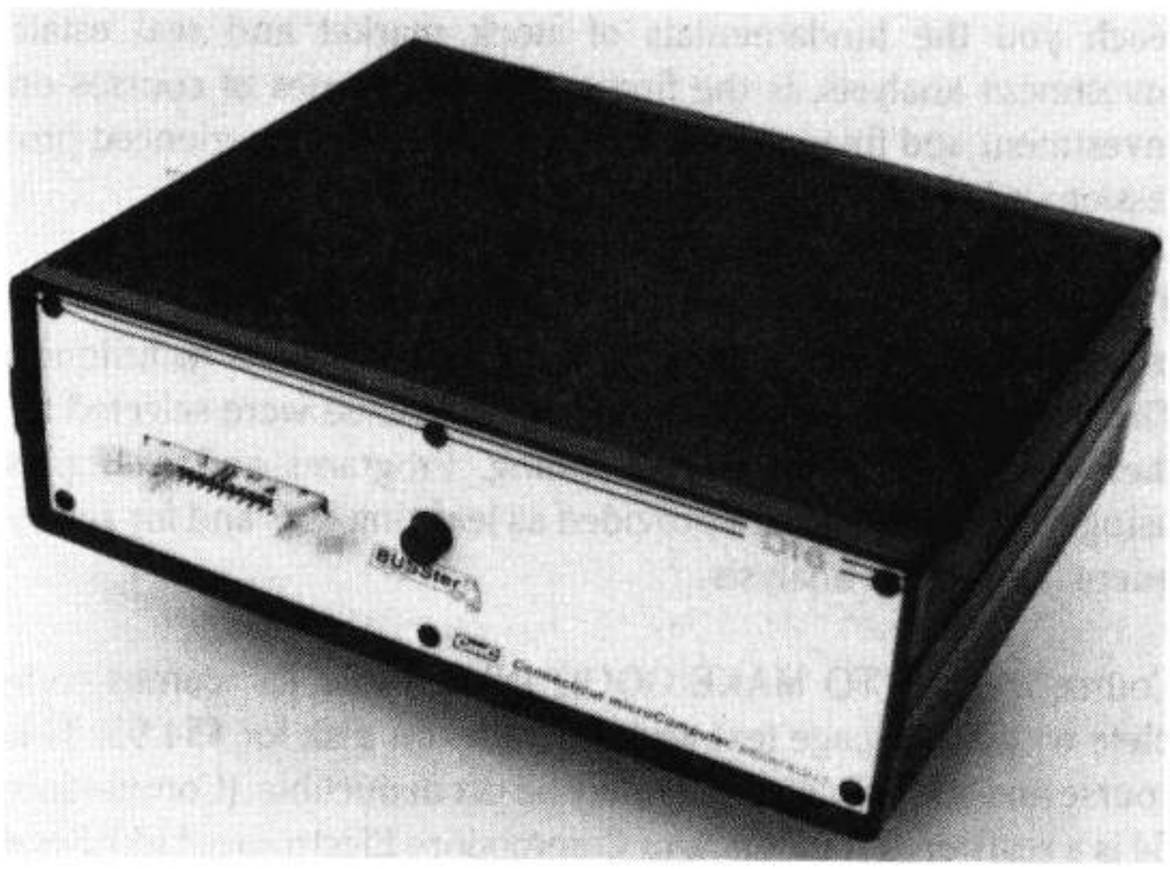
Omnitronix Incorporated
PO Box 43
Mercer Island, WA
98040 206 236 2983

RS–232 16 Channel Analog Input Module

Connecticut microComputer announces a new 16 channel analog input module which is a self contained RS–232 bus compatible device. The USSter D16R works with any computer that has an RS–232 interface (either built–in or added on) including computers manufactured by APPLE, IBM, Commodore, Osborne, Hewlett-Packard, and Tektronix. The D16R is the eleventh product in the BUSSter series of I/O modules.

The BUSSter D16R Analog Input Module accepts commands from any host computer through its RS–232, to read data or activate the timer and buffer. The data is converted to 8 bits (0.25 %) and the conversion time is less than 100 microseconds. The built–in timer operates from .01 seconds to 48 hours. The built–in buffer allows data acquisition while the host computer is busy with other tasks. A BUSSter module economically increases a computer's interfacing capability while reducing its workload.

The BUSSter D16R Analog Input Module is easily programmed through BASIC commands from the controlling computer.



The BUSSter D16R sells for $495.00 in standard version, including case and power supply, and is available from stock.

Shirley Fletcher
Connecticut microComputer
36 Del Mar Drive
Brookfield, CT
06804 203 775 4595

Bits & Pieces

Before we jump into this issue's blitz of random thoughts, is there a difference between a *bit* and a *piece*? Well, how about this: bits are little facts, curiosities, 1 or 2 line programs, POKEs, or SYSes; in other words, quickies. Pieces are presentations of longer programs and contain a paragraph or more of text; sort of like mini–articles. Actually, pieces are often just articles that we couldn't find a home for anywhere else in the magazine. So if you sometimes find yourself skipping over Bits & Pieces to get to the "meat" of the issue, turn back later for dessert!

The Bits Blitz

Built–in debugging aid

Here's an idiosyncrasy that can be put to good use. On a BASIC 4.0 machine (40/8032), performing **SYS 53027** from within a program prints the message " in ", followed by the line number in which the SYS is located. The program then continues normal execution. This is an easy way to trace a recalcitrant program: just insert this SYS at various points in the code, and the messages will show what parts of the program are being executed. In a way, it's more handy than a regular TRACE function, since it only traces the parts of the program you're concerned with. A couple of notes about it: No carriage return is printed after the message, and if you execute it from direct mode, a strange line number is printed out (well, what do you expect?). If it wasn't such a well–kept secret, it would look as though the subroutine was purposely designed as a debugging aid.

Easy Disk Directory Pattern matching

If you want to load a selective directory from a 1541 single disk drive, or from drive 0 of a dual unit, you needn't use the complete syntax:

```
LOAD"$0:pattern",8
```

The command,

```
LOAD"$pattern",8
```

. . .will do the same thing. For example, to see a directory of all programs on drive 0 starting with a "P", just enter:

```
LOAD"$P*",8
```

This leads to any easy way to load just the disk header and number of blocks free:

```
LOAD"$$",8
```

Poison Line Number

Sometimes a computer can get annoyed for the smallest reasons. Enter the following number on your computer (it works with 4032/8032 and 64):

350800

There's actually a whole range of numbers in the same neighborhood that produce the same effect. Try entering it more than once. Why does it happen? Who knows, maybe it's just an unlucky number.

Closing "Forgotten" Files

Editing with Commodore machines is wonderful compared to others, but it can be annoying when all variables are lost whenever a line number is entered, with or without text. Besides clearing variables, though, the machine forgets about all open files. Suppose you OPEN a sequential file to disk and write to it. You MUST close the file afterwards, but if you did any line editing, deliberately or not, the system will think there are no open files, and won't let you close it. Now, you know perfectly well that the file is indeed open, since the light on the disk drive is on.

In such a situation, here are two ways to close the file:
1) The disk drive will automatically close all files when the command channel is closed. To use this feature, just enter:

```
OPEN1,8,15:CLOSE1.
```

ALL open files will then be closed, courtesy of the disk drive.

2) You can change the number of files open in the operating system. This method allows you to close the first file opened, or the first N files opened, rather than all open files like method 1. The change is done with a single POKE:

```
POKE 152,1 on VIC/64
or POKE 174,1 on PET
```

You can then CLOSE the file as usual. If you wish to re–activate more than one old file, change the value of the POKE accordingly.

SAVE–ing a Range of Memory From BASIC

On a 4032 or 8032, you can always save a range of memory from the monitor, for example:

```
S "0:filename",08,8000,8400
```

. . .would save screen memory out to disk. With the C64, such a feature would be even more desirable, so that the picture currently in the high resolution screen could be SAVEd. The 64 doesn't have a built–in monitor like the 4.0 PETs do, but you can SAVE a range of memory by entering a single line from direct mode! Here it is:

sys57812"filename",8:poke193,slo:poke194,shi :poke174,elo:poke175,ehi:sys62954

The variables SLO and SHI are the low and high order of the start address, respectively, and variables ELO and EHI are the low and high end address. (SLO=start AND 255, SHI=start/256, ELO=end AND 255, EHI=end/256)

For example, to save the high resolution screen from 8192 ($2000) to 16192 ($3F40) using the filename "screen" on drive zero, the line would look like this:

```
sys57812"0:screen",8:poke193,0:poke194,32
:poke174,64:poke175,63:sys62954
```

The file can then be LOADed as usual, with:

```
LOAD"filename",8,1.
```

Cassette can be used instead of disk if you change the ",8" to ",1" when saving and loading. Remember, if you're loading the file back in from within a program, you have to make sure it only gets loaded on the first RUN. For example, the first line of the program could be:

```
10 if f=0 then f=1:load"filename",8,1
```

WAIT A SECOND! **Jeff Goebel**

This next BIT is from Jeff Goebel, who writes:

"If you are ever using cassette files on a 64, it is a good idea to first make sure the PLAY button has been shut off. I always include a line:

```
"PRESS STOP ON CASSETTE PLEASE"
```

. . .and a WAIT 1,16 at the beginning of any of my cassette loaded programs. This will STOP the computer until the STOP button on the tape player is pressed. That way, when I later try to read from a file, the PRESS PLAY prompt will again be displayed, and the user has the option to change tapes or whatever, before pressing PLAY. Actually, I can spiff up the standard PRESS PLAY prompt to be almost anything I want by using my own routine. If I include a PRINT statement like;

```
"PRESS PLAY ON TAPE CASSETTE UNIT"
```

. . .followed by a WAIT 1,16,16, the computer will stop and wait till the play is pressed. I then have time to

```
PRINT "THANK YOU"
   or "SEARCHING FOR DATA"
```

. . .before I open the file. Since the play key is already depressed, the computer's own prompt will not appear, and the data will load as normal.

"Actually, the WAITs are universal; WAIT 1,16 will stop until ALL keys are up on the tape unit and WAIT 1,16,16 will stop and wait until ANY key is pressed on the unit. It doesn't have to be the PLAY key specifically."

Checking for SHIFT, CTRL, and Commodore keys

PEEK(653) will yield the state of these three keys; bit 0 for SHIFT, 1 for the "Commodore Key", 2 for CTRL. Study the following example.

```
10 rem* control key demo *
20 print chr$(8): rem* lock case *
30 :
40 for i=0 to 1 step 0
50 ck=peek(653)
60 if ck=0 then print"– none –";
70 if ck and 1 then printtab(1)"SHIFT";
80 if ck and 2 then printtab(8)"Commodore";
90 if ck and 4 then printtab(20)"CTRL";
100 print
110 next i
```

As you can see, the state of any or all keys may be examined with a single POKE, and an AND to see which key(s), if any, are being held down. (Holding down the CTRL key also slows the speed of scrolling).

Changing Screen Character Colours

A quick way to change the colour of ALL characters on the C64 screen:

```
10 c=1: b=53281: rem* c is colour, b is border color reg *
20 s=peek(b):poke b,c:poke 648,160
   :print chr$(147):poke 648,4:poke b,s
```

The above works on 64s with ROM version V2, which sets colour memory to background colour when the screen is cleared. If you have other ROM versions (that set colour memory to character colour), use this line 20:

```
20 poke 646,c:poke 648,160:print chr$(147):poke 648,4
```

The first version won't change the current character colour, it'll just change the colour of all characters on the screen.

It works by telling the operating system that the screen is way up in ROM, so that clearing the screen serves only to set colour memory. The screen page pointer is then set up to its normal default value, 4 (screen at $0400).

Pieces

Death by Garbage

Delays caused by garbage collection (discarding of unwanted strings by the system) are often a minor annoyance, but sometimes uncollected garbage can be the cause of unexpected crashes! Suppose we wanted to write a program to store data from a

sequential file into memory, either to be examined there by a program, or to be written to a new file. The following harmless–looking program should do the trick, right?

```
10 rem* read bytes into memory from seq file *
20 open 8,8,1,"0:lots of data,s,r"
30 bm=4096: rem* start of memory for storage *
40 c=0   : rem* counter *
50 rem— loop —
60 get#8,a$: poke bm+c,asc(a$+chr$(0))
70 c=c+1
80 if st=0 and bm<24576 goto 50
90 rem— endloop —
100 close 8
110 end
```

Bytes are read from the sequential file and POKEd into successive memory locations. The program ends when end–of–file occurs, or memory location 24576 ($6000 in hexadecimal) is reached. When run on a 4032 or 8032, the above program seems to work fine – unless the file is more than about 5000 bytes long. On a long file, the machine will suddenly break into the machine language monitor, or simply halt. Inspection of the data after the crash reveals that it has been totally corrupted. What happened?

It may be obvious to some of you who fully understand the nature of strings in Commodore BASIC, but it may be a surprise to the uninitiated. It occurs because the data being POKEd into memory steps on string storage space. One would think that the 8k of memory between $6000 and $8000 would be more than enough to store the strings; there's just A$, right? Well, the string storage space grows each time a new A$ is read, because a new string is created in memory. Each time a new string is created, the bottom–of–strings pointer decreases (this pointer is at 48–49 in BASIC4, 51–52 in VIC/64). The garbage left over from previous strings won't be collected until this pointer decreases until it equals the top of arrays pointer – in other words, when there's no more free memory. Unfortunately, we want to use the memory between the top of BASIC and variables, and the bottom of strings.

What we really want in the case of the above program is garbage collection after every new byte is read in. That will keep RAM free and safe, since the strings will never grow more than a few bytes (one byte will be required to store A$, and another to store the result of A$+CHR$(0)). The best way to force a garbage collect is by invoking the FRE function. In the above program, we could insert the statement:

```
75 F=FRE(0)
```

This doesn't slow down the program noticeably, since there are only two bytes of garbage to discard each time. In fact, in any program which reads in many bytes of data from disk, or redefines string variables often with GETs or string expressions, it's a good idea to use the FRE function in every iteration of the loop. If the strings are allowed to pile up until a garbage collect is automatically invoked, there could be a long wait in store, especially in BASIC 2.0 machines. A program user may think the machine has crashed during a long garbage collect, and become quite hostile as he turns off the power after waiting ten minutes.

So be careful when POKEing into "free" memory, and use the FRE function liberally in string–intensive programs. As a more drastic measure, CLR will also do the trick, but of course it must be used with care within programs. There's another lesson here: even if a program works fine when tested with relatively small amounts of data, it may die when worked harder or for longer periods of time. By coincidence, if I'm testing a commercial program and it crashes on me, the first word that usually springs to mind, is "GARBAGE".

Drowning in Garbage! — Elizabeth Deal

Liz writes, "We all know about the elegant screen dazzlers. The other end of the computing stick is:"

```
100 rem  drowning in c64 garbage!
110 rem  by elizabeth deal
120 :
130 rem* set top of basic to $4000 *
140 poke55,0:poke56,64:clr:vi=53248
150 :
160 rem* hires screen at $2000 *
170 poke vi+17,peek(vi+17)or32
180 poke vi+22,peek(vi+22)or16
190 poke vi+24,peek(vi+24)or8
200 :
210 rem* define a string 200 times
220 ta=56324: for j=1 to 200
230 v$=chr$(peek(ta))+v$: next j
240 get i$:if i$="" then clr: goto220
250 :
260 rem* exit and restore screen *
270 vi=53248
280 poke vi+17,peek(vi+17)and223
290 poke vi+22,peek(vi+22)and239
300 poke vi+24,peek(vi+24)and247
```

Editor's note:
The wild patterns displayed on the screen are as a result of "garbage" – the string V$ is repeatedly redefined, filling memory, which happens to be video RAM. Far out. **– T.Ed.**

Single Disk Copy Program — Rick Illes, Milton Ontario

The program that follows allows you to make copies of programs, or SEQ files on 2031/1540/1541 disk drives. Files can be of any kind: BASIC, machine language, or sequential. The only limit is the length of the file to be copied, which depends on how much memory your machine has. As it stands, it will work on upgrade and 4.0 BASIC; if you have a VIC 20 or Commodore 64, change these lines:

```
110 poke 828,peek(55): poke 829,peek(56): ds=0
120 poke 55,peek(45): poke 56,peek(46)+1: clr
130 t=peek(55)+256*peek(46): s=t
300 close1: poke 55,peek(828): poke 56,peek(829):clr
```

The program follows:

```
100 rem* single disk copy: by rick illes
110 poke 828,peek(52): poke 829,peek(53)
120 poke 52,peek(42): poke 53,peek(43)+1: clr
130 t=peek(52)+256*peek(53): s=t
140 input"filename";a$
```

```
150 input " Prg or Seq (P/S) " ;t$
160 open 1,8,8,a$ + " , " + t$: if ds goto 300
170 print " ok. . . "
180 rem* read the file in *
190 get#1,b$: poke s,asc(b$ + chr$(.)):s = s + 1
      :if st = . goto190
200 if ds goto 300: rem* disk error (basic 4) *
210 close1: print " qr insert copy disk into drive #0 "
220 print " then press <space> "
230 get b$: if b$ = " " goto 230
240 get b$: if b$<> "   " goto 240
250 open 1,8,8,a$ + " , " + t$ + " ,w " : if ds goto 300
260 print " ok. . . "
270 rem* write the file out *
280 for i = t to s-1: print#1,chr$(peek(i));: next i
290 if ds = 0 then close1: print " q done! "
300 print ds$: close 1: poke 52,peek(828):poke 53,peek(829)
```

Editor's note:
Notice the 'IF ST=.' and 'CHR$(.)' in line 190? This is perfectly acceptable for the value zero. Faster too.

Also note the way that Rick protected memory before storing the bytes from the disk file. He first saves the current top of memory, then sets it to 256 bytes above the top of the BASIC program; that's plenty of space for variables in this case. After the program finishes, it restores the top of memory pointer to their original state. This is a good technique, and it avoids the possibility of "death by garbage", as explained in the piece of the same name. The only thing to watch out for though, is if you want to use the above routine as a subroutine in a larger program: The variables and arrays in that case may need more than the 256 bytes provided above the program. To allocate more variable space, just change the "+1" in line 120 to give more than one 256 byte block. **–T. Ed**

BASIC 4.0 String Bug

Here's a bug, reported by Commodore:

"The bug is a failure to detect 'Out of Memory' error. This can cause corruption of string data or programs if space is running short.

"The bug only occurs in BASIC4 and when there are less than 768 bytes (or 3 times the longest string) free after all variables and arrays have been assigned to a program.

"An example of the bug on a 32k PET:

```
10 DIM A(6330)
20 BUG$ = BUG$ + " W " + " . " : PRINT BUG$: GOTO20
```

"The above program will concatenate a string of alternating characters 'W.W.W.W.W'. The 'Out of memory' terminating is correct but the string is corrupted after only a few passes.

Solution.

"The easiest solution is to trap the 'Out of memory' error from BASIC.

```
IF FRE(0)<768 THEN PRINT " OUT OF MEMORY ERROR " : STOP.
```

Another solution is preventative medicine: Don't concatenate more than two strings at the same time. Doing the concatenation in two steps, ie:

```
BUG$ = BUG$ + " W " : BUG$ = BUG$ + " . "
```

. . .will circumvent the problem.

Intercepting C64 System Error Messages – Elizabeth Deal

By changing the "Error message link" at $0300–$0301 to point to your own routine, you can change the behaviour of the operating system when it prints messages, including "READY.". Your code should jump to the normal error handling routine after it's finished (normally $E38B). The type of error is indicated by the X register; the value $80 (128 decimal) indicates no error, and causes "READY." to be printed.

With that brief explanation, I'll present a few useful applications that were sent in by Elizabeth Deal from Malvern, PA.

(1) If you're tired of seeing ?SYNTAX ERROR you can get rid of the insults: using SUPERMON (or a similar machine language monitor), change the vector at $0300–0301 to point to your code:

```
YOURCODE LDX #$80          ;code for no error
         JMP (SAVEDVEC)    ;back to operating system
SAVEDVEC . . .             ;here goes whatever was previ-
                           ously at $0300/0301 (normally
                           $8b, $e3)
```

This will suppress all error messages, but still print READY.

(2) A slightly more useful thing might be to print all messages at the top of the screen (second line, actually) to prevent scrolling:

```
YOURCODE TXA               ;x holds error #
         BMI OUT           ;no error
         LDA #$13          ;ascii code for " home "
         JSR $FFD2         ;print it
OUT      JMP (SAVEDVEC)    ;remainder of error handling
SAVEDVEC . . .             as above
```

(3) Selective handling of errors can be useful. In graphic situations it is particularly annoying to get a ?FILE NOT FOUND ERROR (#4) as well as a flashing disk light. The light is enough, let's get rid of the error message:

```
YOURCODE CPX #4            ;code for ?file. . . error
         BNE OUT           ;continue if other error
         LDX #$80          ;fake no error
OUT      JMP (SAVEDVEC)
SAVEDVEC . . .             as above
```

(4) you can suppress printing "READY." to avoid messing up the screen. We won't need to save the existing vector here.

```
YOURCODE TXA
         BMI OUT           ;no error
         JMP $A43A         ;normal error with " READY. "
OUT      JMP $A47B         ;skip printing " READY. "
```

A combination of points 3 and 4 could be useful, and the latter point could be modified so that "READY." is only suppressed in certain circumstances.

There is one drawback to suppressing "READY.": any action that doesn't send a final linefeed, such as LIST, will finish with the cursor one line too high. Small price to pay!

(5) This is just a scratch of the surface. The C64 is a programmer's delight, but it can be a nightmare if the housekeeping isn't good. Intercepting the error routine to clean up house (switch out of high-resolution mode, bring back BASIC, restore normal pointers, kill the sprites and so on) permits nightmare-avoidance. Other uses are possible, though I haven't tried them – for instance, how would you like to POKE (address),-40 ? It's a pain to figure out the two's complement value of -40 to feed to some machine language program; using the error vector might help in that one.

Note to Liz: Your hunch was right – we do like this sort of thing.

C64 RESTORE key checking

In last issue's Bits & Pieces, there was a little interrupt-driven machine language program which performed a subroutine whenever a given key was pressed. Well, if you want to use the RESTORE key on the 64, there's an easier way, and it's better: you don't have to change the IRQ vector, so it will work even with IRQ-driven programs.

The RESTORE key is unlike any other key on the keyboard. There is no memory location which can be read to indicate whether or not RESTORE is depressed. Rather, the RESTORE key is connected directly to hardware circuitry which generates an NMI whenever the key is struck sharply (a slow, gentle push won't do the trick).

An NMI, or Non-Maskable Interrupt, is a lot like an IRQ (Interrupt ReQuest), except that it can't be disabled by software. When an NMI occurs, the 64 jumps to the location pointed to by the vector at $0318 and $0319 (792 and 793 decimal) – this vector normally points to $FE47. On the 64, NMIs are used for two purposes: The RESTORE key (to warm-start if the RUNSTOP key is also held down), and for the RS-232 software (an NMI is generated when a character is received on the RS-232 port). The RS-232 routines don't affect detection of the RESTORE key, though. By changing the vector at $0318/9, we can point to our own routine. If this routine transfers control to the usual NMI routine at $FE47 after it's finished, then the interrupt will finish normally and execution will continue from the point where the interrupt occurred.

Detecting the key is incredibly simple. First the NMI vector must be changed to point at our routine. Suppose the routine lives in the cassette buffer, at $033C. The vector could be set up from BASIC like this:

```
POKE 792,60: rem* set nmi low byte to $3c *
POKE 793,3 : rem* . . . high byte to $03   *
```

The code at $033C would perform some action, say, set up certain border and background colours, then JMP to $FE47:

```
033C: A9 00     LDA #0      ;black
033E: 8D 21 D0  STA $D021   ;background
0341: A9 0B     LDA #11     ;dark grey
0343: 8D 20 D0  STA $D020   ;border
0346: 4C 47 FE  JMP $FE47   ;normal nmi entry
```

That's all there is to it. Now, whenever the RESTORE key is struck, the colours will be set up. The normal operation of RESTORE is not hindered, since the normal NMI-handler routine at $FE47 will perform a warm start if the RUNSTOP key is depressed.

A Questionable Prompt

Ok, we all know that BASIC's INPUT statement does us favours and displays a question mark as a prompt, free of charge. Well, sometimes the question mark is totally out of place, since the prompt message isn't a question at all, like: PLEASE ENTER YOUR NAME? –looks a bit silly, doesn't it? To kill the question mark on any machine, without using POKEs or anything machine-specific, open a file to the keyboard (device number 0) as follows:

```
100 open 1,0: rem* open file to keyboard *
110 print " Please enter your name:  ";: input#1,name$
120 close 1
```

Besides killing the question mark, using INPUT in this way does not send a carriage return after entry, so that a message could be printed on the same line as the prompt. Furthermore, you can reject null entry elegantly by adding the line:

```
115 if a$ = "" then 110
```

This makes the prompt seem to ignore a carriage return without text.

While on the subject of opening keyboard files, it should be noted that the real advantage lies in full-screen editing capability. Instead of entering a string in response to the prompt, you can simply move the cursor to any screen line and press RETURN, reading the contents of that line into the INPUT variable. A good application would be user-entry of multiple fields, such as name, address, etc. The user could cursor around to his heart's content, editing the fields to his satisfaction before pressing RETURN over correct fields. The program would read the fields one at a time, and could exit the INPUT loop when a special end string is received, for example, "EXIT".

Fast BASIC HI-RES Point Plot

Here's a short BASIC subroutine which will plot a point on a bit mapped screen. The variable 'B' must be set to point to the beginning of bit mapped screen memory (normally 8192), and the array 'E(' must be initialized with:

```
FOR I = 0 TO 7:E(I) = 2↑(7-I):NEXT I
```

```
1000 rem* plot a point *
1010 I = b + (yand248)*40 + (yand7) + (xand504)
1020 poke I,peek(I)or e(xand7)
1030 return
```

Fast HI-RES Screen Clear

Clearing the bit mapped screen with POKEs from BASIC can be maddeningly slow. Here's a machine language program to zero 8192 bytes anywhere in memory. I'ts 16 bytes long and fully relocatable. The following BASIC program puts the machine language into memory and executes it, clearing the bit mapped screen at 8192 ($2000 in hexadecimal).

```
10 rem* clear hi-res screen *
20 data 162,  32, 160,    0, 152, 145, 251, 200,
        208, 251, 230, 252, 202, 208, 246,  96
30 rem* load ml prog into memory *
30 fori=0to15:read a:poke828+i,a:next
35 :
40 poke251,0: poke252,32: rem* start address *
50 sys828: rem* execute clear routine *
```

Decimal to Hex conversion Table

Brian Dobbs

Before you breath a dec-to-hex-programs-have-been-done-to-death sigh, please note: this one produces a neat looking table on a printer, for future reference. Even if you already have such a table, running this program will save you a run to your nearest photo-copy machine if you need extra copies. As the program stands, it goes from zero to 255, but the top limit can be changed in line 130.

```
100 rem decimal to hex conversion table
110 rem by brian dobbs-timmins, ontario
120 :
130 max=255:rem* highest value in table
140 open4,4,1:x=0:y=1:k$=" --- "
150 print#4,spc(25)" decimal to hex conversion table "
160 :
170 rem— main printing loop —
180 d=x:gosub280: rem* convert to hex *
190 if x>9  then k$=" -- "
200 if x>99 then k$=" - "
210 print#4,tab(5);x;k$;h$;
220 x=x+1:y=y+1:if y=6 then y=1:print#4
230 if x<=max goto180
240 print#4:close4:end
250 rem— end loop —
260 :
270 rem *convert dec to hex subroutine*
280 h$="":d=d/4096:fori=1to4:d%=d:h$=h$+
    chr$(48+d%-(d%>9)*7):d=16*(d-d%):next
290 return
```

Large Characters on VIC or 64

The ROM character generator is accessible to BASIC on the VIC and 64. By PEEKing into the character generator ROM, you can duplicate the shape of all 512 characters in magnified form (8 times larger). The following program asks for the desired character set, and the character to be printed. It uses the subroutine starting at line 330 to print a large image of the character, using asterisks as pixels. The available character sets are:

Char Set	Description
0	upper case/graphics
1	upper/lower case
2	reverse upper/graphics
3	reverse upper/lower

For the VIC version, change line 160 as indicated and delete lines 350, 370, and 440 to end. The 64 version needs extra code because its character ROM is hidden under I/O, and it must be expressly switched in and out. Unfortunately, switching out the I/O will crash the machine unless the interrupts are disabled, so that must be done as well. Subroutines handle the ROM switching.

```
100 rem**********************
110 rem* print large characters  *
120 rem* transactor magazine    *
130 rem* written sep'84  -cz     *
140 rem**********************
150 :
160 crom=13*4096: rem 8*4096 for vic
170 for i=0to7:e(i)=2↑(7-i):next i
180 :
190 print chr$(147)
200 for loop=0 to 1 step 0
210 : input" character set (0-3) ";set
220 : input"            character ";c$
230 : print chr$(147)c$
240 : cp=peek(peek(648)*256)*8
245 : rem 1st screen loc gives char #
250 : rom=crom + set*1024 + cp
260 : print
270 : gosub 330: rem* print image *
280 : print
290 next loop
300 :
310 :
320 -subroutines:
330 rem* translate rom image *
340 for i=0 to 7
350 gosub 460 '* char rom in
360 line=peek(rom+i)
370 gosub 540 '* i/o in
380 for j=0 to 7
390 disp=1:if line and e(j) then disp=2
400 rem* space for 0, '*' for 1 *
410 print mid$("  * ",disp,1);
420 next j:print:next i
430 return
450 :
460 rem* switch char rom in *
470 poke 56334,peek(56334)and 254
480 rem* turn interrupts off *
490 poke1,peek(1)and 251
500 rem* enable character set rom *
510 return
530 :
540 rem* re-enable i/o *
550 poke 1,peek(1)or 4
560 rem* turn interrupts on *
570 poke 56334,peek(56334)or 1
580 rem* switch in i/o *
590 return
```

Letters

Doctor Destructo: I was very pleased with the depth of coverage of software protection in your most recent Transactor on the subject. I'd like to share the following philosophy and machine language routine with you. If you wish you may pass this along through the pages of your magazine.

Quite a number of protection schemes you mentioned used vectors that are, when not used in the specific execution, set to point at system reset ($FCE2 – 64738) in the Commodore 64, which is the computer I own. Some time ago I wrote an educational program composed of 5 program files on disk. The problem of protecting code with a reset is that BASIC or machine language program is not destroyed. A BASIC program can be recovered with an "UNNEW"utility and machine language can be hunted down and copied, the BASIC program being the simplest. I therefore wrote the enclosed code to cover the tracks of my program with binary snow in the event that a reset is performed on it.

I have over 5K of sprites in the program under the BASIC ROM where the VIC II chip can see them but I wanted to destroy those images as well in my reset protection. So I wanted to wipe memory from $0800 to $D000 with $00 bytes and then for the extra milliseconds it takes I copied the I/O and Kernal ROMs to RAM from $D000 to $FFFF. The reason for this is because I think it is possible for a ferret program to be placed in the M.L. buffer or under a ROM chip than can do something devious. The code enclosed emulates a cartridge and is loaded in at $8000 to $8058 so that when a reset is called the start–up codes are checked by ROM routines and then control is transferred over to my program. The program moves itself out of the way into the cassette buffer to avoid being overwritten and therefore bombed, and then zero's out memory from $0800 to $D000. It then copies the ROMs into their underlying RAM from $D000 to $FFFF. If I had continued to fill memory up to $FFFF with zero bytes then the screen would go black for a moment and I didn't want to give hackers any clues at all so I used the ROM to RAM method. The program then calls the hardware reset sequence.

This is all well and good as far as I've taken it. For specific applications such as mine though the code may have to be located elsewhere with the emulation portion (the first 8 bytes) placed before execution. Also there is no stopping someone LOADing a program (the ferret referred to earlier) that sits in an undisturbed area which could wait an appropriate time and then un–lock the program. However, with a small amount of ingenuity, this code could be LOADed by an auto–run boot then be called upon to wipe memory clean before loading in the protection program again and then the program that requires protection. I hope you understand my logic on this.

I chose to transfer the memory wipe out code to the cassette buffer because the reset that is called as the last instruction renews the buffer with zero bytes, so there is little if any evidence of what a hacker is up against when he uses a reset button to break into a program. I hope you enjoy this concept as I certainly did while writing this code.

Chris Jones
Clearbrook, B.C.

**** Reset Protector ****

```
*       =       $8000
.byt $09, $80, $30 ;low byte / high byte start
.byt $60, $c3, $c2 ;code that will emulate
.byt $cd, $38, $30 ;cartridge rom module
;
        cld
        ldy     #$2a    ;set load from address
        sty     $fb     ;hi/lo in zero page
        ldx     #$80
        stx     $fc     ;for indirect set load
        ldy     #$3c    ;to address in zero page
        sty     $fd
        ldx     #$03
        stx     $fe
        ldy     #$00    ;set counter
;
loop    =       *
        lda     ($fb),y ;get a byte
        sta     ($fd),y ;put a byte
        cpy     #$2f    ;last one
        beq     dropout ;if so, drop out
        iny
        bne     loop    ;loop back
;
dropout =       *
        jmp     $033c   ;to new location
        ;
        ldy     #$00    ;set zero page for
        ldx     #$08    ;binary snow job
        stx     $fc
;
snobeg  =       *
        lda     #$00
;
snojob  =       *
        sta     ($fb),y ;and do it
        iny
        bne     snojob  ;loop back
        inc     $fc
        lda     $fc     ;set compare
        cmp     #$d0    ;to end address
        bne     snobeg  ;loop back
        ldy     #$00    ;set up for rom
        sty     $fb     ;to ram transfer
        ldx     #$d0
        stx     $fc
;
romram  =       *
        lda     ($fb),y ;get a byte
        sta     ($fb),y ;copy it
        iny
        bne     romram  ;loop back
        inc     $fc     ;bump address
        lda     $fc     ;set compare
        bne     romram  ;loop back
        jmp     $fce2   ;do a system reset
.end
```

Thanks for the enlightening mixture of thoughts and code. You're right, as most already know, that a regular system reset will clear any previous alterations to operating system RAM, but leave BASIC or user RAM intact. With your code in place, little does the crafty pirate know that troubles are to soon appear on the horizon. Just in case a few of you don't need that much protection, below will be found a bit of deadly code that will leave in its wake little more than binary rubble. It's self modifying, a severe no no, but it purpose is to kill code, not win any prizes. Place it where ever you feel it will inflict the most damage, then point a few vectors at it. It's crude but effective.

```
*       =       $0828       ;or wherever you please
;
        sei                 ;make sure nothing will interrupt us
        ldx     #0
;
death   =       *
        sta     end,x       ;what ever is in the accum is ok
        inx
        bne     death       ;destructo mode !!
        inc     death+2     ;increment the high storage byte
        jmp     death       ;then jump into the pit once again
.end    =       *
```

Coin Side One: Your November editorial, "Piracy: A Fact of Life?" was refreshingly accurate. Every other magazine I have seen caters to the advertisers by taking a strong editorial stand against "piracy" so-called. Yours is the only objective opinion I have read in eight years of computing.

I am the author of "The Code Book", (Loompanics Press, 1979, 1983). My publisher takes the same view you do. He sells copies of my book retail. He keeps the price low enough to discourage photocopying; he also realises that if Person A wants to give Person B a photocopy of the book, there is nothing he can do about it. More importantly, very few buyers would be willing to do this.

The market place puts a limit on software piracy. In days gone by pirates harried ocean trade. Piracy was profitable because Mercantilist tariffs and duties artificially raised the price of imports. Smugglers circumvented these restraints on trade. The (nominal) free trade of the 1700's and 1800's eliminated piracy.

Software piracy is profitable because of "Software Mercantilism". Computers copy data as part of their integral function. Those who attempt to limit the right to are trying to evade the facts of reality. In fact, software companies encourage piracy by placing high price tags on mundane programs. If a commercial package costs $400 and you value your time at $10 per hour, then spending 40 hours overcoming protection schemes is profitable. On the other hand, if the commercial package sells for $150, you have to get the job done in 15 hours to break even. Having spent 15 (or 40) hours at work, who will GIVE AWAY a copy of a pirated program ??

My publisher further discourages copyright infringement by binding the book and putting an attractive cover on it. Documentation is the software equivalent of this. I have tried to use copies of professional software; without the manual, it's easier to write a BASIC routine of my own to do the job then to decipher the syntax of hit-or-miss.

Software piracy will be defeated when the software market enjoys a free trade climate. VisiCalc is worth about $30; Lotus 1-2-3 is worth about $100; WordStar is a $25 value. That's the bottom line on piracy.

Michael E. Marotta
Lansing, MI

Thank you for your comments. And we quite agree - No single force will stop the constant whir of the photocopying machine, nor the duplication of a record album onto a cassette tape. The same holds true for the computer programmer that has a special knack for breaking protection code. The hard line attitudes of some companies regarding copyright infringement, and the ever present threat of legal action, will not determine the true power of copyright. It's the public that will.

Coin Side Two: I received my copy of Volume 5, Issue 03 yesterday and read the letter on page 15 re: Copyright Rights. I found myself frustrated and dismayed that The Transactor could respond this way to such an important issue.

The editorial response missed the point completely. It was hypocritical, without any sense of reasonableness and not well researched.

The Transactor suggests that universal (should be read as "free") access to software should be expected, yet devoted this entire issue to concern over piracy and copy protection; thus implying that programs have some value that should be protected. In addition, I refer to page 2 and The Transactor's copyright policy and suggest that you compare it to that of COMPUTE! magazine.

The original letter ends with an extreme situation that is totally unreasonable. If The Transactor is to remain sympathetic to this "cause", then I would be very interested in what your real life reaction would be to TPUG maintaining one subscription for The Transactor and sending photocopies to about 20,000 members, all in the name of furthering access to educational material. Perhaps COMPUTE! would end up running an article similar to yours on page 7 of this issue which announces the demise of COMMANDER.

I admit that the above situation would be extreme, but is as reasonable as saying that I can't let my four year old daughter use the educational programs that appear in COMPUTE!'s various publications because it is illegal (my subscription, not my daughters). On the other hand, reasonableness does not allow me to provide copies to all my neighbors.

It is not unreasonable to request that teachers contact publishers for permission before attempting to use the material for educational purposes, especially from sources outside of the conventional educational materials. I am certain that requests of this nature would generate positive responses as quickly from COMPUTE! publications as from the Transactor.

Piracy is nothing more than making unauthorized copies of material and distributing them for personal gain, monetary or otherwise. Aside from possible scales of volume, what is the difference between the Apple lawsuits (copyright infringement) and unrestricted copying of program listings published under copyright in magazines. The Transactor published an opinion, in this issue, which flies in the face of unrestricted copying for "educational" purposes.

Also, I am disappointed that The Transactor would make such comments about a direct competitor, especially unprovoked as it is, and not fully researched. The author of the letter did not explain the context in which copyright became a subject for editorial consideration and comment in COMPUTE!'s publications and The Transactor does not appear to have researched it beyond the May 1984 issue.

It goes beyond the use of the printed programs to the wholesale copying of diskettes available from a program entry subscription offered by COMPUTE!'s Gazette. Opinions were requested about a club maintaining one diskette subscription and then providing copies to all club members. The editors were opposed to applying copy protection techniques to these diskettes, but were concerned about protection of the authors' royalty rights. They ultimately decided in favour of the subscribers, at least on an interim basis. Attitudes, as expressed by the author of your letter wanting programs (and possibly services) without charge, will surely aid in changing COMPUTE!'s faith in the sense of fairness of the general public. I expect that sooner or later, some copy protection is going to be introduced because of such misguided opinions.

Personally, I am against copy protection of software because it compromises the purchasers ability to freely use his property. However, I also believe in the rights of the authors and that they should be protected to the extent required. It is the general public that will determine the extent to which copy protection will be required. I am in favour of educating the users of software, but only as far as it is fair to all other parties involved.

I currently subscribe to eight computer magazines including The Transactor and both COMPUTE! publications covering Commodore products. I can attest to the fact that your letter writer is going to lose out as far as gaining knowledge is concerned because I find very little duplication in these magazines with respect to technical matters. It is a shame that it is going to result from such narrow-mindedness.

Aside from this slip in editorial comment and attitude, I find The Transactor to be very interesting and informative, and I eagerly anticipate its arrival well in advance.

James Van Eden
Riverview, N.B.

Your comments have been most influential and well taken. Constructive criticism is as valuable to us as compliments and we invite anyone to send in their beefs.

Without trying to just get the last word in, we would just like to point out that software protection is just another aspect of computing, hence the theme of Volume 5, Issue 03.

As far as COMPUTE!'s policies go, perhaps we did "fly off the handle" prematurely. However, we fully intend to continue stating the facts as we see them as well as reader submissions, no matter how opinionated. The text that originally incited this controversy could have been more carefully worded or the letter from Mr. James would never have been written.

Lastly, it's only natural to copy unprotected disks. If that's how you want to sell the software then the only logical approach is to make the disk so cheap that it becomes uneconomical for the majority (emphasis on majority) to even leave the house to obtain a copy. Even a couple of dollars can make the difference at this point. Like COMPUTE and Ahoy, we too will be offering a diskette service. We fully expect unauthorized copies will be made. But we also expect our price will sell enough of them to make it worth our while. Protected diskettes have only one more safeguard - you must first find someone with a broken copy before you can obtain one. Most don't have such connections, but as long as the price might be considered high enough to warrant some sleuthing, unrestricted copying will continue to be a problem.

Transbloopers

Vol. 5, issue 4
"Dynamic Expression Evaluation"

1)A last-minute typesetting goof-up ended up mutilating the table on page 28 beyond recognition. The table should have looked like this:

PROMPT	RESPONSE
maximum range?	2*π
age in days?	28*365
distance in kilometres?	120*1.6
speed in mph?	90*0.6
yearly revenue?	mtly*12
monthly income?	yrly/12
length of hypotenuse?	sqr(a*a + b*b)

2) In the same article, reference was made to Darren Spruyt's article "in the last issue". This article ("Getting BASIC to Communicate with Your Machine Code") actually appeared in the previous issue: Vol. 5, issue 2 ("The Transition To Machine Language" was the theme).

3) Bits & Pieces: "SHIFTing your WAIT"

A method was given for waiting until the shift key was pressed, but the given application called for halting if the shift key was pressed (just the opposite). The correct statements are:

WAIT 654,1,1 (C64)
WAIT 152,1,1 (40/8032)

The above will cause a halt in a program IF the shift lock key is engaged.

Vol. 5, issue 3
"Picprint"

4)The article made several mentions of using the F1 key to switch between text and high-resolution video modes. The key to use is actually the F7 key, as stated in the source code of the program.

5) Also in the Picprint program, the printer codes used for dual-density graphics are 27 and 121 (ESC y). These codes enable dual-density graphics *at double speed* on the Star Gemini-10X printer. The more common codes for regular dual-density graphics are 27 and 76 (ESC L).

The Commodore DOS: A Review Of Two Books

by David A. Hook
Barrie, Ontario

Title : **The Anatomy of the 1541 Disk Drive**
Authors : Lothar Englisch & Norbert Szczepanowksi
Editors : Greg Dykema and Arnie Lee
Publisher : Abacus Software
PO Box 7211
Grand Rapids, MI 49510
1984, 323 pages
Cost : $19.95 (U.S.)
Audience : advanced, machine language

Title : **Inside Commodore DOS**
Authors : Richard Immers & Gerald Neufeld
Publisher : Datamost Inc.
20660 Nordhoff Street
Chatsworth, CA 91311-6152
1984, 508 pages
Cost : $19.95 (U.S.)
Audience : advanced, machine language

Both these books are recent publications, and deal with the internal disk operating system (DOS) of the Commodore 1541 disk drive. The subject is the same, but the content differs substantially, so it seems appropriate that they be reviewed together.

The **Anatomy** book appears to have been of German origin and is also copyrighted by Data Becker GmbH in Dusseldorf. Like the earlier Anatomy of the Commodore64 it has been translated and published by Abacus. The book has not been typeset, but has been prepared on a letter-quality printer. Of its total content, 151 pages present a disassembly of the 16K ROM inside the drive. This includes the 901229-03 ROM, the "original" disk ROM. The disassembly is nicely laid out, with asterisks and spaces setting off the blocks of code. The book cover indicates that the listing is "fully commented", and this is true of 80% of the disassembly. A large block of code is devoid of comments. The comments are helpful, but do not really fully explain the activity.

The **Inside** book appears to be an outgrowth of earlier work on the Commodore dual drives. I understand that one of the authors, Dick Immers, did his Ph.D. thesis on the subject. Like the other, a large proportion of the book deals with the 16K of ROM. Here we have 205 pages describing the ROM. However, we are not given the actual code, but instead are told what purpose each subroutine fulfills. Reference is made to RAM locations and their purpose, thereby giving a much better sense of what is happening. This part of the book is not typeset, which (to me) means there won't be any typos that the printer introduced. Also, the authors suggest symbolic labels for the various routines, a handy touch. The documentation is also that of the earlier ROM, but the text identifies the few changes made for the 901229-05 ROM.

To many, the above content would be the reason to buy the books. We should examine what other topics are covered in each:

Anatomy gives a review of the disk commands, to supplement the information in the (terrible) Commodore manual. To develop familiarity with sequential files, a BASIC mail list program is developed and explained. The concepts of relative access are explained and a BASIC home accounting program serves to apply the techniques. Chapter 2 documents the syntax and purpose of the direct-access commands. Brief example programs demonstrate their use. The technical information in Chapter 3 shows a block diagram of memory allocation and gives the purpose of the low memory "variable" locations. A brief picture of how the data is organized on the disk surface gives hexadecimal dumps of several "types" of disk sectors. Chapter 4 gives quite a number of helpful utilities for dealing with the disk drive. Many are in machine language, and the assembler source code is supplied. The most sophisticated of these is a "diskmonitor" which allows you to read, display, edit and write a sector directly.

Inside gives a thorough description of the principal DOS commands. Chapters 3 and 4 detail the organization of the data on the disk. More than 40 pages are devoted to presenting diagrams of the data on the sectors where BAM, directory, program and all types of file entries are recorded. Chapter 5 introduces the direct-access commands. The example programs are extensive, and fully explained. Some could qualify as full-fledged utility programs in their own right. Chapter 6 is really meaty—how the floppy disk controller actually executes a DOS command. This is a preface to Chapter 7, where we learn the encoding scheme for the data written on the disk. The authors present thirteen programs that permit you to duplicate DOS-protected diskettes. Error-writing schemes are given for all the known methods of copy-protection. The assembler source code for the ML portions is given. These programs really show you how the disk works. Chapter 8 has very welcome advice on recovery from trouble. A number of utility programs is supplied. A comprehensive overview of the 1541DOS is given in Chapter 9. The major routines of both the Floppy Disk Controller and the Interface Processor are identified, and how to use them. Details of the timing of the read and write cycles are explained. The chapter concludes with a discussion of DOS bugs, both real and imagined, and some opinions on the "write-incompatibility" of the 1541 and the 4040 type drives.

I really liked the **Anatomy**'s discussion of relative and sequential files. The example programs are quite useful and instruct quite well. Abacus uncovered an "M" mode for reading an unclosed file, and also found a way to "scratch-protect" files. These have been deep, dark secrets unknown to almost all users. The lack of more detailed analysis of the ROMs is a comparative shortcoming. (However, for a long time, no one had documented as much as they had.)

The **Inside** book is a remarkable piece of scholarship. It ranks right alongside Raeto West's **Programming the PET/CBM** in its comprehensive treatment of the topic. I find that the description of the ROMs is more useful than the actual code, particularly given the quality of the information. The book is highly controversial in its treatment of backing-up of copy-protected disks. One expert suggests that the level of copy-protection used with Commodore format disks is pretty unsophisticated at present anyway. After stating that I'd never buy software that I couldn't back-up, I have a scad of such disks now. In my view, it's not improper to want a copy **on-hand**, rather than waiting weeks for that replacement to arrive from the manufacturer. The 46 utility programs are a real boon. The wealth of information in this book makes it very easy to recommend to the advanced user. The book is well-written, and treats each topic thoroughly. Moreover, the language is plain English and quite readable.

I have purchased both books. **The Anatomy of the 1541** predates the other, but I do not regret having invested in it. Both books are recommended to you, but if you choose to buy only one, then **Inside Commodore DOS** should be the choice.

Machine Language For The Commodore 64 And Other Commodore Computers: A Review

by David A. Hook
Barrie, Ontario

Author : Jim Butterfield
Publisher : Brady Communications Co., Inc.
A Prentice–Hall Publishing Co.
BOWIE, MD 20715, U. S. A.
1984, 326 pages
Price : $12.95 US ($17.95 Canadian)
Audience : Beginner, machine language programming

Beginners who wish to venture into the world of 6502 (6510) machine language have many more choices nowadays. We veterans of PETdom have waded through some pretty muddy waters to learn some of the tricks. There is a swarm of books on the subject today, but it's hard to find a single one that admits that the microprocessor does not operate in a void. The chip lives inside a real computer, and that means the environment must be considered when you are talking about a real machine language program. An earlier text is the Levanthal book (1). The examples there caused the PET to run away and hide—the programs had been placed in a very sensitive area of memory!

There is another major factor. A television game show called Jeopardy (revived after eight years), provides the "answer", while asking contestants to provide the "question". Let's play:

The Answer: Jim Butterfield
The Question: What is missing from all other books on machine language programming?

Mr. Butterfield is recognized as the world's foremost authority on Commodore computers. Few people have not been touched by his work as a writer, and as Associate Editor of COMPUTE! since its inception. We in the Toronto area have been particularly blessed, since we have seen Jim perform as a teacher. His talent is impressive—he has a knack of explaining the most difficult concepts in a simple and entertaining way. The book is written in this light conversational style.

The microprocessor in the Commodore 64 is the 6510, which has the same set of instructions as the MOSTechnology6502. Jim Butterfield co–authored a book of machine language programs for the 6502 (2) way back in 1978.

The combination of Jim's writing skill and his practical approach to the topic are unbeatable. The best expression of the theme of the book is that stated by the author in the Preface. Programmers learn by doing, so give lots of examples and projects for them to do. To enter and use the examples, you must have some tools that are machine–specific. Although these have almost nothing to do with the machine language, you are helpless without them. So Jim provides the tools and tells you how to use them. Also, you must understand the architecture of the computer—your programs have to fit somewhere in memory and you need to know how to perform input and output operations.

He has stuck to those principles throughout the book, and there is an excellent reason for this. More than five years ago, Jim started a course in machine language for beginners. Over that period it has been refined, and now exists as a two–day intensive program. Jim has taken his show on the road, and thousands of people have participated in his seminars—all over the world. With that background, it's easy to understand why the book is well–organized and unified in theme.

The book is divided into eight lessons with a reference section that represents more than half the total pages. We'll discuss that part later.

Chapter 1 discusses the first concepts. We meet the microprocessor, the address and data busses, cover binary and hexadecimal notation, and the internal registers. On the practical side, we learn about a machine language monitor (MLM). The MLMs for Commodore have a common set of commands, so these are explained.

The monitor is described here because we have a real example to enter and execute. Each chapter concludes with a review and the problem projects.

Chapter 2 is titled Controlling Output. We must rely on some built-in subroutines if we expect to get any output. These are "kernal" routines, and it's explained how we use CHROUT. Up till now, we did "hand assembly" of the ML program. Now we see the assembler/disassembler functions of the "extended" monitors. Jim emphasizes that we could have done this from the start, but maybe we can appreciate what is really going on, by doing it the hard way first. The magical SYS instruction allows us to link ML with BASIC—it's not so mysterious after all.

Chapter 3 deals with the flags in the status register, logical operations and the kernal input (GETIN) and stop-key test routine. Jim insists that these flags have been given names that are confusing and only half-right. So, he suggests new names for them—the "Zero" flag should be called the "Equals" flag. Again, misconceptions are headed off before they can be entrenched. The interpretations of "signed numbers" and "overflow" are made clear.

Chapter 4 involves numbers and arithmetic. Numbers larger than one byte are described and then how to work with them with add, subtract, multiply and divide. This is done lightly, and the depth won't scare the beginner away. No attempt is made to explain floating-point arithmetic, as Jim balances the flavour of the subject against the potential confusion. Using subroutines for modular programming is described.

We are now half way through the book, and there has not been any attempt to classify the instructions into address modes. Other authors tend to throw all 56 instructions and their 13 modes at you right away. Since you haven't got a clue what they're all about, you might give up. Now the address modes mean something, so Chapter5 becomes the right time to discuss them. As usual, the descriptions are very clear. But the illustrations are much less useful than they should be. Jim gives concrete numeric examples in the text part, but the diagrams just show arrows, with no values. This is much below the high standards elsewhere in the book. I also feel that the "bug" in the 6502/6510 should have been mentioned: the indirect jump instruction will fail if the indirect address sits just below a page boundary.

Chapter 6 covers the problem of where to place your ML program. This material alone justifies buying the book. No other ML book gives you such an insight into the computer's organization. The native BASIC interpreter forces you to know all the implications that can affect the coexistence with ML. How BASIC stores its programs and variables, and manipulates its pointers is very important. Passing values between BASIC and ML is included.

In Chapter 7, we learn about the stack, the USR function and two advanced techniques: the interrupt and the wedge. What the interface adapter chips do is also included. If you want to add commands to BASIC, your ML program must "wedge" itself into the operating system. The CHRGET subroutine of the machine is used to fetch the next character from BASIC. It is explained in detail. Then an example program demonstrates how to infiltrate BASIC.

In the final chapter, another "difficult" topic is covered. We can do input from the keyboard and output to the screen. Other input/output devices are part of the system—how do we connect these? Jim describes three more ROM routines that make the job a snap. The treatment of tape, disk and printer is entirely consistent with the default devices (screen and keyboard)—so we merely apply what has already been learned. Lots of unnecessary hair-pulling and sleepless nights will be prevented. A few brief hints on debugging are given. Using the mini-assembler has been an important aspect of the training, so he wraps up by describing what a symbolic assembler can do. Again, now that we know how things really work, it's OK to use a more advanced tool.

Because the appendices are huge (170 pages), only a list is given here. Microprocessor instruction set, machine characteristics of all Commodore generations, memory maps of "low" memory including zero-page availability and associated chip functional maps (all PET versions, VIC, Commodore64, Plus/4 & BSeries), ROM detail maps for the C64, a superchart of Commodore and ASCII codes, exercise answers for VIC and PET users, BASIC loader for Supermon64, and complete manufacturer's references to the interface chips (6520, 6522, 6525, 6526, 6545, 6560, 6566 & 6581).

The final appendix refers to a disk with a variety of utilities and demo programs. Jim says that it will be available shortly, as an optional purchase. There is brief documentation for the programs on the disk.

Well, now for the hard part: who should buy this book? Jim states quite clearly that it is a book for the beginner in machine language programming. I submit that no other book I've seen does this job properly. So, if you wish to venture into ML, this is the one. The teaching of a ML course calls for a textbook—you could hardly do better.

People who are familiar with the fundamentals can still gain insight from it. There are a few projects there that are worthwhile exercises for the intermediate programmer. You will not find the key info, regarding the architecture of Commodore equipment, so clearly expressed elsewhere. And the appendices are a gold mine of reference data.

Now for the hot shot programmer, who really has a grasp of the content and the computer. Maybe that reference info is all you'll find, but it might be real handy to have alongside your machine. And how many of you experts field a constant stream of questions on ML? With this on your bookshelf, you could readily recommend it to those beginners who are hounding you.

Don't wait for the second printing. Get a copy (or two) right away.

References:

1. **6502 Assembly Language Programming** by Lance A. Levanthal, OSBORNE/McGraw-Hill, Inc. (1979).

2. **The First Book of KIM** by Jim Butterfield, Stan Ockers and Eric Rehnke, Hayden (1978).

Commdore 16/Plus–4 Memory Maps

Jim Butterfield
Toronto, Ont.

I'm happy to see that Commodore have (I think for the first time) made an early publication of "official" RAM memory maps for their new machines. You can find them in the November/December 1984 issues of "Commodore Microcomputers" magazine, and they include something quite valuable: Commodore's internal "labels" by which they identify the locations. With these labels, an assembly language programmer can now use standard identifiers in writing a program – I'll certainly try to do so.

I've been picking apart the logic of the machines, and I'd like to offer my maps, too. They are similar in wording to previous maps I've published.

A few things that are noticeable about the new machines. They have a new architecture which calls for you to read the map more carefully.

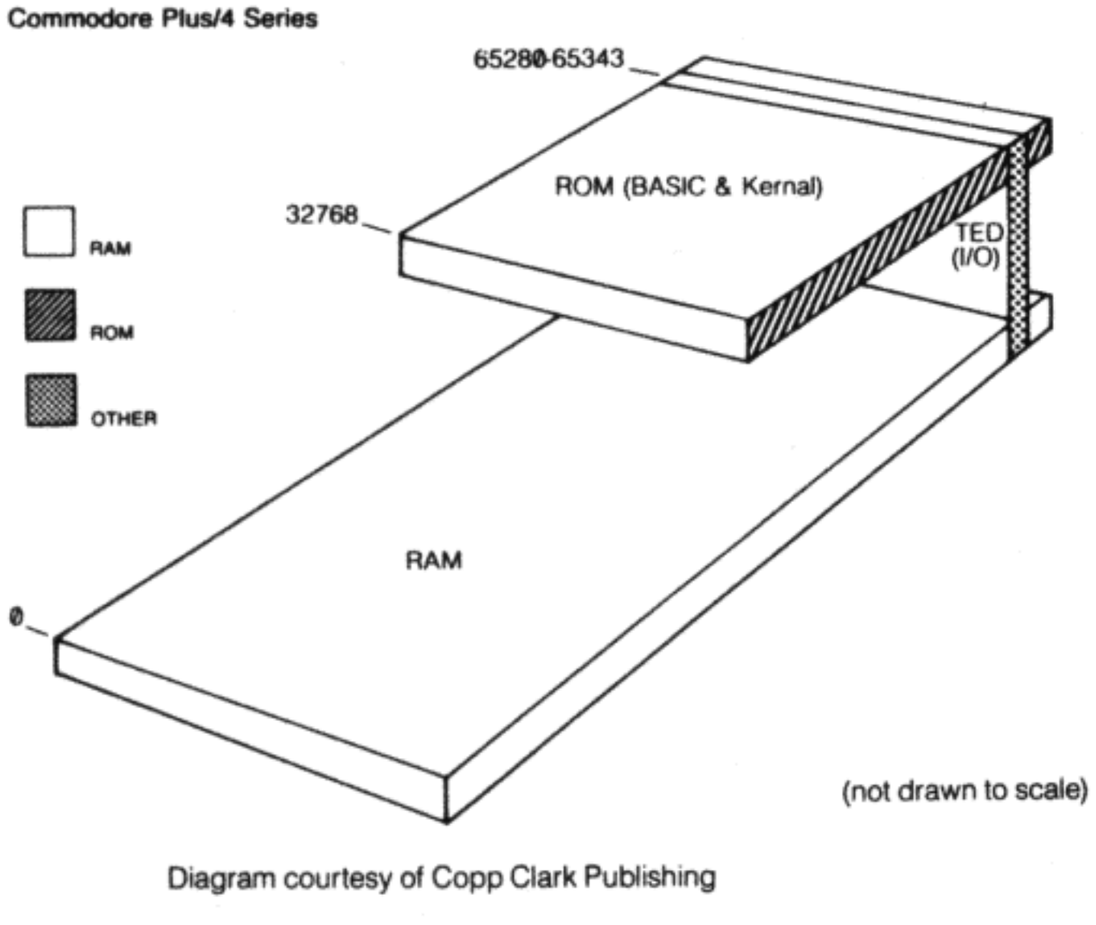


Diagram courtesy of Copp Clark Publishing

Since RAM lies partly beneath the ROM or ROMs above, we must understand that an address might refer to any of several "levels". For example: PRINTPEEK(32768) will give you a value from RAM at that location; but if you switch to the monitor and display the contents of hex 8000, you'll see the ROM and get quite a different value. For most applications (and for memory locations below 32768), you won't need to worry about all this. But when you get into the advanced stuff, you'll need to know how to work all the picky details.

A Few Differences

Inner space engineers will notice that much of zero page looks very much like that of the Commodore 64. The first big surprise is probably the CHRGET routine: it's missing from zero page. It turns out that CHRGET (which has relocated to page 4) needs to be used in a more complex way, since the calling routines must specify if they are looking for information from ROM or RAM.

And wonder of wonders: There's a whole block of spare memory in zero page, from at least $D8 to $E8. It's enough to disorient a programmer.

Extra space is put to a variety of uses. Page 1 is still mostly the system stack (see the note about the Basic stack, below). Page 2 is input areas as before, with space for working the new DOS commands. Page 3 contains links and vectors similar to those on the 64; watch these closely, since the absence of an NMI shortens up the table by one address.

Page 4 contains some communications buffering, and some replacements for the missing CHRGET routine. There's also a work area for PRINT USING and other activities. The current definitions of the programmable function keys take up much of page 5, and page 6 seems to be largely reserved for system expansion, such as speech synthesis.

There's a new stack mostly in page 7, the Basic stack, which holds loop and subroutine type information. FOR/NEXT, GOSUB/RETURN, and DO/LOOP with its WHILE and UNTIL provisions. It lets you write somewhat more complex programs, and leaves the machine stack less cluttered.

The screen is now accompanied by a sucessor to the "color nybble" table; it's called the "attribute" table. It's like the color nybbles, but contains extra information: not just color, but hue as well, and also a "flash" bit. It's in main memory now, residing at hex 0800 (decimal 2048), with the screen following it at hex 0C00 (decimal 3072). The extra space means that Basic starts higher than before, at hex 1000 or decimal 4096.

The ROM system is quite massive. At 32K, a map ends up lengthy even if it's abridged. A few surprises include: a built–in machine language monitor; graphics, error trapping and disk commands built into Basic; and a clever means of internally cataloging all the ROMs that happen to be fitted and making them available as desired.

The same old Kernal routines are still there and do the same job, but there's new stuff, including an "unofficial Jump Table" to handle some clever bank switching tasks.

It looks like a lot of fun. Good hunting. . ...

Commodore 16 / Plus 4 RAM Memory Map

(Preliminary: September 25/84. Note that the previously available locations for VIC/C64, $00FC to $00FF, are now used and are not available.

Hex	Decimal	Description
0000	0	Chip directional register
0001	1	Chip I/O; serial bus/cassette
0002	2	Loop type match
0003–0006	3–6	Renumber parameters
0007	7	Search character
0008	8	Scan–quotes flag
0009	9	TAB column save
000A	10	0 = LOAD, 1 = VERIFY
000B	11	Input buffer pointer / # of subscripts
000C	12	Default DIM flag
000D	13	Type: FF = string; 00 = numeric
000E	14	Type: 80 = integer; 00 = floating point
000F	15	DATA scan/LIST quote/memory flag
0010	16	Subscript/FNx flag
0011	17	0 = INPUT;$40 = GET;$98 = READ
0012	18	ATN sign/Comparison evaluation flag
0013	19	Current I/O prompt flag
0014–0015	20–21	Integer value
0016	22	**Pointer:** temporary string stack
0017–0018	23–24	Last temporary string vector
0019–0021	25–33	Stack for temporary strings
0022–0025	34–37	Utility pointer area
0026–002A	38–42	Product area for multiplication
002B–002C	43–44	**Pointer:** Start–of–BASIC
002D–002E	45–46	**Pointer:** Start–of–variables
002F–0030	47–48	**Pointer:** Start–of–arrays
0031–0032	49–50	**Pointer:** End–of–arrays
0033–0034	51–52	**Pointer:** String–storage (moving down)
0035–0036	53–54	Utility string pointer
0037–0038	55–56	**Pointer:** Limit–of–Memory
0039–003A	57–58	Current BASIC line number
003B–003C	59–60	Textpointer: BASIC work point
003D–003E	61–62	**Pointer:** BASIC stack for CONT
003F–0040	63–64	Current DATA line number
0041–0042	65–66	Current DATA address
0043–0044	67–68	Input vector
0045–0046	69–70	Current variable name
0047–0048	71–72	Current variable address
0049–004A	73–74	Variable pointer for FOR/NEXT
004B–004C	75–76	Y–save; op–save; BASIC pointer save
004D	77	Comparison symbol accumulator
004E–0053	78–83	Misc. work area, pointers, and so on
0054–0056	84–86	Jump vector for functions
0057–0060	87–96	Miscellaneous numeric work area
0061	97	Accum#1: exponent
0062–0065	98–101	Accum#1: mantissa
0066	102	Accum#1: sign
0067	103	Series evaluation constant pointer
0068	104	Accum#1 hi–order (overflow)
0069–006E	105–110	Accum#2: exponent, and so on
006F	111	Sign comparison, Acc#1 versus #2
0070	112	Accum#1 lo–order (rounding)
0071–0072	113–114	Cassette buffer len/Series pointer
0073–0074	115–116	Auto line number increment
0075	117	Graphics flag
0076–0079	118–123	Misc work values
007C–007D	124–125	BASIC pseudo–stack pointer
007E–008F	126–143	Misc work values
0090	144	Status word ST
0091	145	Keyswitch IA: STOP and RVS flags
0094	148	Serial output: deferred character flag
0095	149	Serial deferred character
0096	150	Register save
0097	151	How many open files
0098	152	Input device, normally 0
0099	153	Output CMD device, normally 3
009A	154	Direct = $80/RUN = 0 output control
009B–009C	155–156	**Pointer:** tape buffer, scrolling
009D–009E	157–158	End of program pointer
009F–00A0	159–160	Work area
00A1–00A2	160–161	Monitor working vector
00A3–00A5	163–165	Jiffy Clock HML
00A6	166	Serial bit count/EOI flag
00A7	167	Tape shift byte
00A8	168	Serial cycle count
00A9	169	Temporary color vector
00AA	170	Countdown,tape write/bit count
00AB	171	Number of characters in file name
00AC	172	Current logical file
00AD	173	Current secondary address
00AE	174	Current device
00AF–00B0	175–176	Pointer to file name
00B1	177	Tape error count
00B2–00B3	178–179	I/O start address
00B3–00B4	180–181	Load address pointer
00B6–00B7	182–183	**Pointer:** start of tape buffer
00B8–00B9	184–185	Misc. pointer
00BA–00BB	186–187	Cassette I/O work pointer
00BC–00C1	188–193	Work pointers
00C2	194	Screen reverse flag
00C3	195	End–of–line for input pointer
00C4–00C5	196–197	Input cursor log (row, column)
00C6	198	Which key: 64 if no key
00C7	199	Input from screen/from keyboard
00C8–00C9	200–201	Pointer to screen line
00CA	202	Position of cursor on above line
00CB	203	0 = direct cursor; else programmed
00CC	204	Current screen line length
00CD	205	Row where cursor lives
00CE	206	Last I/O character
00CF	207	Number of INSERTs outstanding
00D0–00D7	208–215	Unused; reserved for speech
00D8–00E8	216–232	Unused
00E9	233	Work value
00EA–00EB	234–235	Color line pointer
00EC–00EE	236–238	Screen work values
00EF	239	Number of characters in keyboard buffer
00F0	240	Screen freeze flag
00F1–F4	241–244	Monitor work values
00F5	245	Cassette checksum
00F6	246	Monitor work value
00F7–00F8	247–248	Cassette work values
00F9	249	DMA control mask
00FA	250	Work byte
00FB	251	Current ROM bank
0100–01FF	256–511	Processor stack area
0200–0258	512–600	BASIC input buffer
0259–025A	601–602	Previous Basic line number
025B–025C	603–604	**Pointer:** Basic statement for CONT
025D–02AC	605–684	DOS command work area
02AD–02B0	685–688	Graphics cursor, X and Y
02B1–02B4	689–692	Graphics working cursor
02B5–02CB	693–715	Graphics work area
02CC–02E8	716–744	Print–using, graphics work area
02E9	745	Temp screen row number
02EA	746	String length
02EB	747	255 = Trace on
02EC–02EE	748–750	Directory work area
02EF	751	Graphics work area
02F0	752	Number of graphics parameters
02F1	753	Parameter relative (1) or absolute (0)
02F2–02F3	754–755	Float–fixed vector
02F4–02F5	756–757	Fixed–float vector
02F6–02FD	758–765	Unused
02FE–02FF	766–767	Reserved for cartridge vector
0300–0301	768–769	Error message link [8686]
0302–0303	770–771	BASIC warm start link [8712]
0304–0305	772–773	Crunch BASIC tokens link [8956]
0306–0307	774–775	Print tokens link [8B6E]
0308–0309	776–777	Start new BASIC code link [8BD6]
030A–030B	778–779	Get arithmetic element link [9417]
030C–030D	780–781	Crunch hook vector [896A]
030E–030F	782–783	List hook vector [8B88]
0310–0311	784–785	Execute hook vector [8C8B]
0312–0313	786–787	Interrupt link (CE42)
0314–0315	788–789	IRQ vector (CE0E)
0316–0317	790–791	Break interrupt vector (F44C)
0318–0319	792–793	OPEN vector (EF53)
031A–031B	794–795	CLOSE vector (EE5D)
031C–031D	796–797	Set–input vector (ED18)
031E–031F	798–799	Set–output vector (ED60)
0320–0321	800–801	Restore I/O vector (EF0C)
0322–0323	802–803	Input vector (EBE8)
0324–0325	804–805	Output vector (EC4B)
0326–0327	806–807	Test–STOP vector (F265)
0328–0329	808–809	GET vector (EBD9)
032A–032B	810–811	Abort I/O vector (EF08)
032C–032D	812–813	USR vector (F44C)
032E–032F	814–815	LOAD vector (F04A)
0330–0331	816–817	SAVE vector (F1A4)
0332–03F2	818–1010	Cassette buffer
03F3–03F6	1011–1014	Tape write/read counters
03F7–0436	1015–1078	RS232 input buffer
0437–0472	1079–1138	Tape error log
0473	1139	CHRGET subroutine
0479	1145	CHRGOT subroutine
0494	1172	Subroutine (self banking)
04A5	1189	Subroutine (bank via $3B)
04B0	1200	Subroutine (bank via $22)
04BB	1211	Subroutine (bank via $24)
04C6	1222	Subroutine (bank via $6F)
04D1	1233	Subroutine (bank via $5F)
04DC	1244	Subroutine (bank via $64)
04E7–04EA	1255–1258	PU characters (,.$)
04EB–04EE	1259–1262	String work area
04EF–04F6	1263–1270	TRAP and error flags
04F7	1271	Stack pointer for error trap
04F8–04FB	1272–1275	DO loop work area
04FC–04FF	1276–1279	Sound work area
0500–0502	1280–1282	USR program jump
0503–0508	1283–1288	RND seed value
0509–0512	1289–1298	Logical file table
0513–051C	1299–1308	Device number table
051D–0526	1309–1318	Secondary address table
0527–0530	1319–1328	Keyboard buffer
0531–0532	1329–1330	Start of BASIC memory
0533–0534	1331–1332	Top of BASIC memory
0535–0536	1333–1334	Timeout/end flags, not used much
0537–0538	1335–1336	Tape buffer counts, not used much
0539	1337	Tape buffer pointer
053A	1338	Tape file type
053B	1339	Character (color) attribute
053C	1340	Flash flag
053D	1341	Unused
053E	1342	Screen page (unused)
053F	1343	Keyboard buffer size
0540	1344	Key repeat: 128 = all, 64 = none
0541–0542	1345–1346	Key repeat counters
0543	1347	Key shift flag
0544	1348	Key font interlock flag
0545–0546	1349–1350	Key input vector (DB7A)
0547	1351	Text/Graphics mode lockout flag
0548	1352	Scroll enable flag
0549–054A	1353–1354	Screen work values
054B–0551	1355–1372	MLM work locations
0552–0557	1362–1367	MLM registers (PC/SR/A/X/Y)
0558–055C	1368–1372	MLM work locations
055D	1373	FN key pending count
055E	1374	FN key pointer
055F–05E6	1375–1510	Key definition area
05E7–05EB	1511–1515	DMA work locations
05EC–05EF	1516–1519	ROM ID (PAT) table
05F0–05F1	1520	Long Jump vector
05F2–05F4	1522–1524	Long Jump registers
05F5–06EB	1524–1791	Reserved RAM for extra ROMs
06EC–07AF	1792–1967	BASIC pseudo–stack
07B0–07CC	1968–1996	Tape working values
07CD–07D0	1997–2000	RS232 working values
07D1	2001	RS232 in pointer
07D2	2002	RS232 read pointer
07D3	2003	RS232 input counter
07D4–07D8	2004–2008	RS232 work values
07D9–07E4	2009–2020	Character load program
07E5	2021	Current screen bottom margin
07E6	2022	Current screen top margin
07E7	2023	Current screen left margin
07E8	2024	Current screen right margin
07E9	2025	0 = Scrolling enablled
07EA	2026	255 = Auto Insert enabled
07EB	2027	Previous character printed
07EC–07ED	2028–2029	Current (color) attribute
07EE–07F1	2030–2033	Screen line wrap table
07F2	2034	SYS A–reg save
07F3	2035	SYS X–reg save
07F4	2036	SYS Y–reg save
07F5	2037	SYS status reg save
07F6	2038	New key detect
07F7	2039	Lockout Ctrl–S
07F8	2040	Monitor read: ROM or RAM
07F9	2041	Color decode switch
07FA	2042	Split screen bit mask
07FB	2043	Split screen video base
07FC	2044	Tape motor interlock
0800–0BE7	2048–3047	Color memory
0C00–0FE7	3072–4071	Screen memory
1000–FFFF	4096–65535	BASIC RAM memory (normal)
2000–FFFF	8192–65535	BASIC RAM memory (hi–res)
8000–FFFF	32768–65535	ROM: BASIC
D000–D7FF	53248–55295	Character sets in ROM
FD00–FD0F	64768–64783	ACIA communications chip
FD10–FD1F	64784–64799	Parallel port/6529
FDD0–FDDF	64976–64991	ROM bank select (write only)
FE00–FEFF	65024–65279	DMA disk interface
FF00–FF1F	65280–65311	TED I/O control chip
FF3E–FF3F	65342–65343	ROM/RAM select (write only)

Commodore 16 / Plus 4 ROM Memory Map

8000 C-16 ROM start
8003 Warm start
8019 Basic setup
802A Fix/float vectors
802E Intialize Basic
80BC CHRGET pointers
80C2 Print Basic start msg
8105 Page 3 vectors
8123 CHRGET copy
818E Keywords
8383 Action vectors
8415 Function vectors
8453 Dfunct vectors
8471 Messages
866F Print 'READY.'
8683 Error routine
870F Ready for Basic
872E Handle new line
8818 Rechain lines
885A Receive input line
8871 Scan Basic-stack
8905 Expand Basic-stack
8953 Crunch tokens
8A3D Find Basic line
8A79 Perform [NEW]
8A93 Run
8A98 Perform [CLR]
8AED PUDEF characters
8AF1 Back up text pointer
8AFF Perform [LIST]
8BBC Perform [RUN]
8C9A Perform [RESTORE]
8CD8 Perform [STOP]
8CDA Perform [END]
8D03 Perform [CONT]
8D2C Perform [GOSUB]
8D4D Perform [GOTO]
8D83 Perform [RETURN]
8DB0 Perform [DATA]
8DBE Scan for next statement
8DC1 Scan for next line
8DE1 Perform [IF]
8E0B Perform [REM/ELSE]
8E1B Perform [ON]
8E3E Get fixed point number
8E7C Perform [LET]
8FE0 Perform [PRINT#]
8FE6 Perform [CMD]
9000 Perform [PRINT]
9088 Print from (y.a)
90A6 Print format char
90B8 Perform [GET]
90EE Perform [INPUT#]
9108 Perform [INPUT]
9142 Prompt and input
914F Perform [READ]
9294 Perform [NEXT]
9314 Check type match
932C Evaluate expression
9471 Fixed-float conversion
9485 Eval within parens
94AD Search for variable
95F8 Evaluate <OR>
95FB Evaluate <AND>
9628 Evaluate <COMPARE>
969B Perform [DIM]
96A5 Locate variable
973A Check alphabetic
9744 Create variable
985B Array pointer subroutine
9871 Float-fixed conversion
989B Set up array
9A2F Compute array size
9A62 Evaluate <FRE>
9A76 Fixed-float
9A7D Evaluate <POS>
9A86 Check direct
9A9D Perform [DEF]
9ACB Check FN syntax
9ADE Perform [FN]
9B54 Set up string descriptor
9B66 Evaluate <STR$>
9B70 Calculate string vector
9B74 Set up string
9BDA Concatenate
9C1B Build string into memory
9C4B Discard unwanted string
9C52 Make room for string
9CAA Clean descriptor stack
9CBB Evaluate <CHR$>
9CCF Evaluate <LEFT$>
9D03 Evaluate <RIGHT$>
9D15 Evaluate <MID$>
9D46 Pull string params
9D61 Evaluate <LEN>
9D67 Exit string mode
9D70 Evaluate <ASC>
9D81 Input byte parameter
9D93 Evaluate <VAL>
9DD2 Get params for POKE/WAIT
9DDE Get params for SOUND
9DE4 Convert to fixed point
9DFA Evaluate <PEEK>
9E12 Perform [POKE]
9E1B Evaluate <DEC>
9E6A Perform [WAIT]
9E87 Evaluate <subtract>
9E9E Evaluate <add>
9F7B Complement FAC#1
9FB7 Multiply by zero byte
A01E Evaluate <LOG>
A07B Evaluate <multiply>
A0A9 Multiply a bit
A0DC Memory to FAC#2
A107 Memory to FAC#2
A137 Adjust FAC#1/#2
A154 Under/overflow
A162 Multiply by ten
A183 Divide by ten
A197 Evaluate <divide>
A21F Memory to FAC#1
A24C FAC#1 to memory
A281 FAC#2 to FAC#1
A291 FAC#1 to FAC#2
A2A0 Round FAC#1
A2B0 Get sign
A2BE Evaluate <SGN>
A2CE Fixed-float
A2DD Evaluate <ABS>
A2E0 Compare FAC#1 to memory
A327 Float-fixed
A358 Evaluate <INT>
A37F String to FAC#1
A453 Print 'IN..'
A45A Print number
A46F Float to ASCII
A5E4 Evaluate <SQR>
A5EE Evaluate <power>
A627 Evaluate <negative>
A660 Evaluate <EXP>
A6B3 Series evaluation 1
A6C9 Series evaluation 2
A707 Evaluate <RND>
A760 Save Basic-stack
A769 Restore Basic-stack
A772 Trim Basic-stack
A77D Kernal calls
A7B5 Perform [SYS]
A7CF SYS return
A7DE Perform [SAVE]
A7F0 Perform [VERIFY]
A7F3 Perform [LOAD]
A84D Perform [OPEN]
A85A Perform [CLOSE]
A86B Params for LOAD/SAVE
A89D Check default parameters
A8A5 Check comma
A8A8 Params for OPEN/CLOSE
A906 Allocate string space
A954 Garbage collection
AA57 Calculate end of string
AA70 Evaluate <COS>
AA77 Evaluate <SIN>
AAC0 Evaluate <TAN>
AB1A Evaluate <ATN>
AB8F Perform [RENUMBER]
ADCA Perform [FOR]
AE5A Perform [DELETE]
AEF7 Print using
B42B Perform [TRAP]
B440 Perform [RESUME]
B4BE Evaluate <ERR$>
B507 Evaluate <HEX$>
B544 Perform [PUDEF]
B557 Perform [DO]
B5AC Perform [EXIT]
B603 Perform [LOOP]
B652 Perform [TRON]
B655 Perform [TROFF]
B6CD Perform [AUTO]
B6E8 Perform [HELP]
B729 Perform [KEY]
B849 Perform [SOUND]
B8BD Perform [VOL]
B8D1 Perform [PAINT]
B9D4 Perform [CHAR]
BAE2 Perform [BOX]
BD35 Perform [GSHAPE]
BE29 Perform [SSHAPE]
BF79 Evaluate <RGR>
BF85 Evaluate <RCLR>
BF87 Evaluate <RLUM>
BFC1 Evaluate <JOY>
BFFD Evaluate <RDOT>
C01E Perform [CIRCLE]
C37B Set graphics cursor
C3F7 Parse graphics command
C48F Get graphics parameter
C4D9 Perform [DRAW]
C50F Perform [LOCATE]
C51A Perform [COLOR]
C567 Perform [SCNCLR]
C5B8 Perform [SCALE]
C5C3 Perform [GRAPHIC]
C7BF Confirm graphics
C8BC Perform [DIRECTORY]
C941 Perform [DSAVE]
C951 Perform [DLOAD]
C968 Perform [HEADER]
C99C Perform [SCRATCH]
C9CC Perform [COLLECT]
C9DA Perform [COPY]
C9F4 Perform [RENAME]
CA00 Perform [BACKUP]
CB1F Parse DOS command
CE00 Interrupt entry
CE0E IRQ sequence
CE60 Do screen split
CEF0 Kernal - UDTIM
CF26 Kernal - RDTIM
CF2D Kernal - SETTIM
CF8A Get color mode
CF96 Fetch memory
CFBF Handle tape motor
D000 Graphic character set
D400 Text character set
D802 Screen addresses
D834 Kernal - SCREEN
D839 Kernal - PLOT
D888 ESC-n normal screen
D8A1 Setup screen line
D9BA Quote test
D9C7 Screen output wrap
D9D9 Setup screen print
DB11 Kernal - SCNKEY
DC41 Function keys
DC49 Output to screen
DC9B ESC-O; key escape
DE06 Decode escapes
DE1A ESC vectors
DE48 ESC-R; reduce screen
DE5E ESC-T; top window
DE60 ESC-B; bottom window
DE8B ESC-I; insert line
DEA0 ESC-D; delete line
DECB ESC-Q; erase to end
DEE1 ESC-P; erase fm start
DEF6 ESC-V; scroll up
DF04 ESC-W; scroll down
DF1D ESC-L; scroll enable
DF20 ESC-M; scroll disable
DF26 ESC-C; cancel insert
DF29 ESC-A; auto insert
DF39 Check screen line wrap
DF46 Break screen wrap
DF59 Make screen wrap
DF66 Calculate screen wrap mask
DF82 ESC-J; start-line
DF95 ESC-K; end-line
E01E Keyboard sets
E153 Send 'Talk'
E156 Send 'Listen'
E181 Send to serial bus
E1E9 Serial timeouts
E1F7 Send listen SA
E1FC Slear ATN
E203 Send talk SA
E20C Wait for clock
E21D Send serial deferred
E22F Send 'Untalk'
E2B8 Serial clock on
E2BF Serial clock off
E2C6 Serial output '1'
E2CD Serial output '0'
E2D4 Get serial & clock
E2DC Delay 1 ms
E319 Print 'Press play & rec'
E31B Print 'Press play'
E38D Start tape
E3B0 Kill motor
E3B7 Clear tape buffer
E3C3 Setup tape buffer
E413 Send tape cycle
E447 Send tape 'long'
E452 Send tape 'short'
E45D Send tape 'medium'
E468 Send tape '0' bit
E474 Send tape '1' bit
E48C Send tape byte
E535 Initiate tape write
E56C Write tape header
E68E Bit masks
E9CC Find any tape header
EA21 Find specific header
EA5B RS-232 out (IRQ)
EA95 RS-232 in (IRQ)
EBD9 Kernal - GETIN
EBE8 Kernal - CHRIN
EC0E Get from tape
EC14 Get from RS-232
EC1C Get from serial
EC4B Kernal - CHROUT
EC63 Send to tape
EC84 Send to RS-232
EC8B Kernal - ACPTR
ECDF Kernal - CIOUT
ED18 Kernal - CHKIN
ED60 Kernal - CHKOUT
EDFA Kernal - TALK
EE1A Kernal - TKSA
EE2C Kernal - LISTEN
EE4D Kernal - SECOND
EE5D Kernal - CLOSE
EF08 Kernal - CLALL
EF0C Kernal - CLRCHN
EF23 Kernal - UNLSN
EF3B Kernal - UNTLK
EF53 Kernal - OPEN
F005 Send SA
F043 Kernal - LOAD
F064 Load from serial
F0F0 Load from tape
F172 Print filename
F194 Kernal - SAVE
F1A4 * Save link *
F1B5 Save to serial
F228 Print 'SAVING'
F234 Save to tape
F265 Kernal - STOP
F2A4 System reset
F2CE Transfer page 3 vectors
F2EB Vectors page 3
F352 Identify 16K/32K/64K RAM
F3D2 Key lengths
F3DA Key definitions
F40C Kernal - SETNAM
F413 Kernal - SETLFS
F41A Kernal - SETMSG
F41C Kernal - READST
F41E Change ST
F423 Kernal - SETTMO
F427 Kernal - MEMTOP
F42F Set MEMTOP
F436 Kernal - MEMBOT -
F445 Perform [MONITOR]
F44C BRK/USR entry
F478 Perform [.R]
F4D7 Perform [.M]
F50A Perform [change reg]
F529 Perform [.>]
F54B Perform [.G]
F570 Monitor commands
F580 Monitor vectors
F5CE Perform [.C]
F5D1 Perform [.T]
F60E Perform [.H]
F66E Perform [.S/.L/.V]
F70A Perform [.F]
F724 Perform [.D]
F83D Op code mode
F881 Machine language codes
F89B Mnemonics
F91F Perform [.A]
FB72 Decrement $F1/2
FB86 Decrement $9F/A0
FB94 Increment $A1/2
FBB7 Save registers
FBC1 Recall registers
FC19 Kernal - IOBASE -
FC59 'Phoenix' routine
FC7F Long Fetch routine
FC89 Long Jump routine
FCB3 IRQ entry
FCB8 Long IRQ routine
FCF1 'SRT' kernal entry
FCF4 'Phoenix' entry
FCF7 Long Fetch entry
FCFA Long Jump entry
FCFD Long IRQ entry
FF90 Jump table
FFFC System vectors

The MANAGER Column

Don Bell
Scotland, Ontario

Letters to The Manager

Joseph J. Maus of Water Mill New York has been working on a 39 week moving average and having trouble with the report generate part of it. I'm not quite sure what the application is – but it sounds like a spreadsheet might be a better way out. If anybody out there has a solution, please let me know.

There are as always a feast of letters decrying the manual, particularly the REPORT GENERATE and the ARITHMETIC options. All I can say is try to get a hold of the articles I wrote on these options and read the column for further pointers.

Richard E.C. Holm of Oakville, Ontario does his Christmas cards with THE MANAGER. He doesn't like the tacky look of mailing labels (I don't either). Thus, he prefers to print each name/address directly on an envelope. The problem is that REPORT GENERATE will not pause during a printed report so that he can insert a new envelope. A solution to this problem is presented below.

Help For Mail Lists

I have had a question on how to get a mailing list report to print only one name/address at a time. This would allow for non–form feed printers; also, it's sometimes difficult to get the labels to print exactly at the right position.

Unfortunately, THE MANAGER REPORT GENERATE has no sophisticated 'print pause' feature. A way around this problem is to output the report to disk as a sequential file. Then use a wordprocessor that can read sequential files created by other programs. Once you have the data available to a word processor, it is it is usually no trouble to specify a pause at the end of each page, or in this case, a name/address label.

The only other way around this problem within THE MANAGER program itself that I know of is to simply run a separate report for each mailing label. You need a field that gives a unique number for each record (you may want to use a field for record numbers). Then you can run the report specifying only 1 particular record. When you want the next label, run the report again only change the search criteria to the number of the next record you want printed. This is very tedious work!

The Easy Way of Revising a File – Changing Prompts

In a previous article I discussed revising a file using the REARRANGE function in the MANIPULATE FILES option. Frankly, I've had mixed success with this operation. I would try to avoid making substantial revisions to your file. If you think it may need revisions, create only a small file and enter only a few records. Thus, if you decide to revise your file and start from scratch again, you won't have wasted much time in data entry.

Hindsight is wonderful if you can still prevent it from being "too late" to act on. One way to revise your file without really revising it is to simply change the prompt for one of the fields you're not using very often. If you overdesigned your file (as most of us do) you probably have 1 or 2 fields that are expendable or at worst a luxury. If you feel that the new piece of information is more important than one of your present fields, then it seems like a fair trade. The important point is that the field type be the same (i.e. numeric or alphanumeric) and the length of the field be long enough to accommodate the new information. You can then revise the screen in the CREATE/REVISE option. Just watch out for that deadly prompt:

FILE HAS BEEN ALTERED. NEW FILE(Y/N)?

A 'Y' ANSWER TO THIS QUESTION WILL DESTROY YOUR EXISTING FILE.

Speeding Up Data Entry – Change the First Field Entered

When you want to update a file, you'll often find that the only field you wish to change is near or at the bottom of the

screen. To make an entry in that field you will have to skip down through a number of fields tp get there. What a pain! The '64 MANAGER has a neat way around this problem.

You may have noticed on the bottom of the ENTER/EDIT screen an 'F' as one of the menu choices. The 'F' stands for first field, i.e. the first field the cursor goes to when you are in the ENTER/EDIT mode. Normally, the cursor is placed in field 1 first. However, the cursor can start in any field you wish, even the last one. To select the 'F' function:

(1) Clear the ENTER/EDIT screen (Shift CLR/HOME);
(2) Press 'F';
(3) Enter the field number you wish the cursor to start at;
(4) Press <RETURN>.

Now when you change a record, enter a new record, perform a search or accumulate, the cursor will begin in the field you have chosen as your 'first field'. After making an entry in this field, the cursor will jump (down or across) to the following field. Unfortunately, there is no way of changing the order of entry of the following fields. Thus, you can only make a jump on the first field entry. This feature is particularly handy if you are only making one change to several records. If you had just revised one of the field prompts in your file as I described above, then this would be a very useful tool.

For the Fastest Record Lookup in the West – Use an Indexed Search

When I first started using the '64 MANAGER I was very impressed with how fast it could search through a big pile of records and quickly find the one I wanted. Of course, 'fast' was relative to my fumbling through an address book or pile of papers. If you thought that was 'fast', wait till you try an indexed search – "swoosh" it's on the screen almost immediately!

Why is an indexed search so much faster? It's much faster to read from memory than it is to read from disk.

In a normal 'search', the computer immediately starts reading the disk, sequentially looking at every record to see if it fits the search criteria. When it finds the right record it brings a copy of it into memory and displays it on the screen. A lot of time is wasted reading the disk while looking for the right record.

If you know which field is the field you are most likely to conduct a search on, then you can define it as your 'key field' and create an index for this field. (Common key fields are surname, part#, invoice# and date.) An index is a list or table in the computer's memory. This index in memory contains the key field for every record. With an 'index search' the computer scans the key field in memory first, quickly finding the required record and its exact location on disk. Then the computer zaps out to the disk, gets the required record and displays it on the screen. It only reads from the disk for a very short period of time when getting the record. Unlike the 'normal search', no time is waste looking for the record on disk.

To create an 'index' on any field for fast lookups:

(1) Make sure you are in ENTER/EDIT option;
(2) Clear your screen – Shift CLR/HOME;
(3) Press Shift I;
(4) Enter the field number you wish to create an index on;
(5) Press <RETURN>.

To perform an 'index search':

(1) Press I;
(2) Enter the word, name, number, etc. you are looking for in the key field (the cursor should already be there);
(3) Press 'backarrow' key.

The index is in ascending 'alpha' order. You can view the previous or next entry in the table by pressing '+' or '–'.

There is no reason why you have to maintain the same index file all the time. It can be changed to suit your particular lookup needs. You can only have one index field at any one time. But if you are doing a lot of searching on a particular field, it will probably save you time to create a new index, perform your searches, and re-create the previous index for the 'key field'.

DON'T PHONE WRITE!

If you have questions about the '64 MANAGER, or would like to submit your own application, please write me a legible, coherent letter. If you submit an application be sure to send screen dumps of ENTER/EDIT screens, reports, math, and sample data. Write to:

Don Bell,
PO Box 23
Scotland, Ontario
N0E 1R0

I will answer letters either directly or through this column. Subscription enquiries should be addressed to The Transactor, 500 Steeles Ave., Milton, Ontario. Canada, L9T 3P7.

Subroutine Eliminators

Jeff Goebel
Georgetown, Ont.

As always, I have brought forth a few goodies that do the work of entire subroutines. This month, most of the article is dedicated to one powerful poke that opens up a world of new unexplored subroutine eliminators.

Before starting off, I must apologize to any people who trusted last issue's UN–NEW poke for the 64. (poke2050,1:sys42291) It does indeed bring back your listing after a NEW or sys64738 but unfortunately, it does not clear up all your pointers. It you attempt to re–edit lines or enter new lines. . .WHAMMO! A bizarre crash the likes of which almost deserves a spot in BITS & PIECES. It does however allow you to LIST the program fine, so you could LIST it to disk and create a sequential file which could later be converted back into a program. Anyway, I have since discovered that the POKE works well with machine language routines (which start with a BASIC SYS command) but not well with pure BASIC. Sorry about that. I hope nobody erased a month's work testing it!

Before the main attraction, I should explain the POKE I left you with in the "Spiffy Listings" article in Volume 5, Issue 3. The POKE caused line numbers to vanish from program listings, but I didn't explain what it was good for. Well, the idea is, it's any easy way to create a nice neat sequential file. The file can then be sent over a modem, used from within a program, printed out (on a printer), or loaded into many common word processors. To create the file, start off by entering a line number followed by a quote and your text. The format will resemble a program, but you won't actually be using any commands as such. It should end up looking something like the example below:

```
10 " Hello there people. I am a short
20 " piece of text. In a moment, I
30 " will become a sequential file.
```

When you have finished typing in your entire file, enter:

```
      POKE 22,35  for VIC/64
or:   POKE 19,32 for CBM BASIC 2.0/4.0
```

If you LIST your program now, you'll see that the line numbers have temporarily vanished. It is now a simple matter to open up a file to the disk or printer and LIST your program to it. To dump to disk, enter OPEN 1,8,1, " FILENAME,S,W " :CMD1:LIST (RETURN), then PRINT#1:CLOSE1 To print it, OPEN4,4:CMD4:LIST (RETURN), then PRINT#4:CLOSE4 when the listing has finished.

The trick is even better if you have a BASIC AID type utility (like POWER) in memory. Using the extra features combined with an extension like that makes this POKE quite a nice word processor, complete with scrolling up and down, search and replace, auto line numbering and renumbering and extra text space. Not bad for one poke. It leaves the realm of subroutine elimination and enters the area of complete program elimination.

The rest of this article describes one all–powerful POKE I've never seen documented. There have been many articles written about self modifying programs using the dynamic keyboard. Here, I'd like to present a SELF MODIFYING POKE. Basic commands are stored in memory as tokens. Each command has a corresponding value. If we were to search through our BASIC memory with a machine language monitor, we would see all the BASIC tokens stored along with our regular text. Since BASIC is stored in RAM, we can POKE to it. It logically follows that our BASIC tokens can easily be changed by poking a new value over top of the old value. Enter and RUN the example below.

```
10 rem 64738
20 poke 2053, 158: rem* change first token to "SYS" *
```

(For machines other than VIC 20 or C64, use the value 1029 instead of 2053 in the above and all subsequent examples. Use 3077 for the +4/C16)

When the program is RUN, the poke in line 20 will place the token value 158 over whatever was there before. It will change line 10 from a harmless REM, to a machine resetting SYS command. That is because 2053 is the first location of our BASIC program, and the value 158 is the BASIC token value of the SYS command. If you now list the program, line 10 will read: 10 sys 64738. The next time this example is run, the computer will reset. You may feel this is quite useless, and I would tend to agree. However, it demonstrates the concept well. In the following examples, we poke some of the other BASIC commands into our first line of BASIC; some of them have more useful applications.

```
10 if x = 0 then x = 1 : load " machine language " ,8,1
20 poke 2053, 143 : rem* change first token to " REM " *
```

With this poke, the program will load a subroutine only the first time. The next time it is run, line 10 will have been changed to a REM statement and therefore the statement will be ignored. This is much easier than waiting for your subroutines to re–load every time you run a program.

```
10 rem4,4 : cmd4
20 poke 2053, 159 : rem* change 1st token to " OPEN " *
```

When the above example is run the first time, all will function

normally. When it is run a second time, all output will be transferred to the printer instead of the screen. The rem will now be an OPEN command.

```
10 rem1000
20 poke 2053, 137 : rem* change 1st token to "GOTO" *
```

This application – perhaps one of the most handy – will change your program so that when it runs the second time, it is capable of bypassing all the opening routines and jumping directly to the main program. The value 137 changes the REM to a GOTO. Most of us know how dull it is to sit through a graphic opening twenty times. Substitute the 1000 value in line 10 for the statement that your main program starts with.

```
10 poke 53280, 0 : poke 53281, 0 : poke 2053, 143
```

In most of the programs I write, I change the screen and border colours to be black, but I give the user the opportunity to re–set the screen to what ever combination is their favorite. When line 10 is re–run, it will not re–poke the screen black. The POKE will have been changed to a REM.

```
10 rem10, 0, 70, 87, 76, 67, 10
20 poke 2053, 131 : rem* change 1st token to "DATA" *
```

The above example changes the REM in line 10 to a DATA statement so when the program is run the second time, a new series of values will be READ.

```
10 rema = 1000:b = 1000
20 poke 2053, 136: rem* change 1st token to "LET" *
```

The above example will change the rem in line 10 to be the LET command. Normally the LET command is rather useless, and you may have never used it before, but here, it allows the variables in line 10 to be active on the second run only. This produce different results each time the program is run.

These are a very few examples of the power of this poke. Refer to the TOKEN CHART at the end of this article and experiment on your own. However, there is also another way to use this same concept. We can poke values in that are NOT tokenized keywords. By doing this, we are able to change other aspects of our Basic statement. Enter the example below.

```
10 a = 1000
20 poke 2053, 66: rem* start 1st line with "b" *
```

Here, we are pokeing the value of the character "b" to location 1. This will change 10 to read b = 1000. Since there are no tokenized keywords here, it is easy to calculate the position of our change. Location 2053 is the first character; location 2054 is the second; 2055 the third and so on. If we know this, we can change b = 1000 to b = 2000 or b = 2001 simply by poking the values for 2 or 1 in the correct positions.

To someone with enough imagination and logic, these concepts open up a new world. It now becomes fairly simple to create a BASIC program that has the power to delete spaces from within a program, or change ALL REMs to POKES, or change all occurences of one character to another, similar to a word processor's search and replace function.

TOKEN LOOKUP TABLE

(most common Basic commands are included – for a complete list, see The Reference Transactor Vol. 4 Issue 5)

abs	182	let	136
and	175	list	155
asc	198	load	147
chr$(	199	mid$	202
close	160	new	162
clr	156	on	145
cmd	157	open	159
cont	154	peek	194
data	131	poke	151
def fn	150,165	print	153
dim	134	read	135
end	128	rem	143
for	129	right$	201
next	130	run	138
step	169	save	148
to	164	stop	144
get	161	sys	158
goto	137	val	197
if	139	wait	146
input	133	verify	149
int	181		
left$	200		
len	195		

Editor's note: While Jeff's technique of having a program dynamically modify itself as it runs is clever and thought–provoking, extensive use of it should probably be avoided, especially in programs of any complexity. Self–modifying code means that the instructions contained in a program at any given point depend on what parts of that program were executed before you look at it. What a debugging nightmare! Used in moderation, as in Jeff's examples, the technique can be quite handy. Note that the above examples only change a single token (the first one in the program, at 2053), and the change is only made at one point in the program, always on the second line. Adhering to this practice, and documenting the effect of the change with a REM statement on the second line should avoid creating uncontrollable, chameleonic monster–code.

Furthermore, the technique shown changes the original token to a new token, but never replaces the original. To alternate between two token, say LET and REM, change the second line to:

```
10 rema = 1000:b = 1000
20 poke 2053, 279–peek(2053)
```

The number 279 is the sum of the token values for REM and LET. One the first RUN, the token in line 10 is changed to LET, so that one the second RUN the variables A and B will be set. For the third RUN, line 10 will again start with REM. To alternate between any two tokens, you must start out with one or the other (since the POKE is dependent of the previous value) and replace the number 279 in line 20 with the sum of the two token values (original plus alternate). **–T. Ed.**

Introducing TransBASIC

Nick Sullivan
Scarborough, Ontario

trans–, *pref.* 1. Across, beyond, on or to the other side, through, into a different state or place. 2. Of, or pertaining to, The Transactor.

The version of BASIC built into the Commodore 64 has come in for a lot of criticism, and it isn't hard to see why. Most 64 owners use a disk drive, yet BASIC does not support the disk commands provided with some other Commodore computers. Most of us were drawn to the machine in large part by its powerful graphics and sound capabilities, yet commands to take advantage of them are not included. Like a treasure chest without a key the 64 demands in exchange for its secrets a lot more work than we ever expected.

There are remedies.

One is to get along, grumbling, with the BASIC they gave us, using PRINT# to send commands to the disk, PEEKing and POKEing the hundred or so memory locations that are relevant to sound and graphics programming. The cost? Style, size and speed — speed above all. Many graphics and sound applications would be undermined entirely if they had to depend on the tools BASIC provides.

Another remedy is to use machine language subroutines. The advantages of machine language are many — conciseness, blistering speed, unparalleled control over the computer. Not only that, but once you know how, writing machine language is fun. Alas, a lot of people do not know how, and lack the time and inclination to learn. And even if you can write machine language, or have other access to the utilities you need, the interface with BASIC is rarely elegant and often problematical.

The third choice is to acquire an enhancements package — Simon's BASIC, for example, which boasts an enormous set of additional commands (more than 100) in several categories. That's dandy, but again there are drawbacks, lack of portability being probably the most serious. That kind of package also tends to be inflexible: if you want only half a dozen of those 100 commands, tough. You can't just pick the ones you want and use the remaining memory for something else — it's all or nothing. And if by chance you want a command the package does not include, that too is your misfortune.

That's the story so far. Now read on.

Starting with this issue The Transactor is offering another approach to expanding Commodore 64 BASIC, one designed to avoid the problems outlined above. The new approach is called 'TransBASIC'. It, too, will offer a large number of commands — eight in this first instalment; in the long run perhaps 500 or more. The commands will include both statements and functions, and they will look and feel just like the BASIC commands we're used to. Unlike other packages however, TransBASIC will leave it up to you which extra commands you add from those currently available. This means that you won't be wasting memory on unused commands, and that in many instances your customized package will be easily transportable via DATA statements.

Is there a catch? Well, yes.

TransBASIC is written in assembly language source code, specifically that of Pro Line Software's PAL assembler, by Brad Templeton. Unless you have access to a copy of PAL, or some other assembler that parasitizes the BASIC editor, TransBASIC is not for you.

Obviously, dependence on PAL is a severe restriction. On the other hand, PAL has a well–earned reputation for its small size, ease of use, and truly exceptional speed, so I don't mind suggesting that if you have any interest at all in working with machine language, you probably should own PAL anyway.

How TransBASIC works

If you aren't much interested in the nuts and bolts of TransBASIC, I invite you to skip this section or return to it later. You can use TransBASIC perfectly well without knowing the technical details given here.

TransBASIC is built around a kernel of less than 500 bytes, which is mostly devoted to four fundamental routines. These routines are entered through the link vectors provided by BASIC at hex addresses $0304 through $030B.

Crunch Tokens: The first of the four routines is responsible for tokenizing the new commands. The fact that TransBASIC tokenizes each input line before BASIC itself does has some interesting consequences.

One is that TransBASIC can't make use of BASIC–style program tokens (hex values $80 through $FF), since BASIC's tokenizer throws out any character in this range. TransBASIC therefore uses a two byte token. The first byte is the normally unused left–arrow character (ASCII $5F), and the second byte,

which identifies the keyword, is a character in the ASCII range $40 through $7F. The only troublesome character in that range is $5E, the up–arrow, used by BASIC as the exponentiation operator and therefore not available for use as a token. The four kernel routines take an extra step to deal with this character.

Doing the TransBASIC tokenization first also means that it is possible for TransBASIC keywords to include BASIC keywords within them, but not the other way around. SORT, for instance, would be a legitimate TransBASIC keyword, even though it contains the BASIC keyword OR, because the 'OR' would be subsumed into a TransBASIC token before BASIC had a chance to see it. STORE, however, would not be a good TransBASIC keyword because it would disable BASIC's RESTORE command: RESTORE would become RE plus a TransBASIC token. A side effect is that if you are used to abbreviating BASIC keywords (e.g. pE for PEEK) you may find that TransBASIC has intruded to the extent of commandeering some abbreviations to keywords of its own (e.g. eX for EXP becomes eX for EXIT).

Expand Token: The second routine in the TransBASIC kernel takes care of expanding the tokens back into keywords for LIST. Fascinating fact: the BASIC keyword list contains exactly 255 characters plus a zero byte. This is maximum capacity — the list is scanned using indirect indexed addressing, in which the index is a one–byte value. The same restriction applies to TransBASIC's own keyword list; therefore in implementing a large TransBASIC application, with many keywords, the total number of keyword characters has to be considered.

Execute Statement: The third kernel routine is responsible for executing TransBASIC statements. (In this context, the word 'statement' refers to those commands that, like PRINT, POKE, LOAD, etc, may be used as verbs in a BASIC instruction.) This routine is fairly straightforward. One complication, though, is that the IF statement in BASIC does not employ the execute link–vector when executing the clause that follows THEN; instead it jumps across the vector directly into the BASIC ROM. Naturally, the ROM knows nothing about TransBASIC keywords and tends to identify them as SYNTAX ERRORS. To get around this difficulty the IF statement is redefined in the TransBASIC kernel (and as a bonus allows an often–handy ELSE clause).

Evaluate Expression Element: The fourth kernel routine takes care of TransBASIC functions ('functions' are commands that, like PEEK and ABS, are used within BASIC expressions, and return a value). TransBASIC permits a variety of function types.

Apart from these four crucial routines, the kernel contains a short initialization section, the IF and ELSE statements described above, and an EXIT statement, which disables TransBASIC and restores the old values of the four link vectors.

Typing in the programs

The TransBASIC system, as we have seen, consists of a number of files written in PAL source code. The system makes it simple to add a selection of new commands to Commodore 64 BASIC, resulting in a TransBASIC dialect tailored to a particular application. The source code for the dialect is generated in two stages. First, the TransBASIC kernel (Program 1 following this article) is loaded from disk into BASIC memory space with the command:

```
LOAD "TB/KERNEL",8
```

Second, modules containing routines for added commands are merged with the kernel, and line 95 of the combined program is updated (as described later) to reflect the total number of added statements and functions in the dialect. The merging of modules with the kernel is most easily done using the TransBASIC ADD statement, which you will have available to you after typing in the programs that accompany this article.

If you want to avoid the hassle of typing in the programs, of course, they can also be found on The Transactor's program disk for this issue. In that case you can skip from here directly to the next section, Using TransBASIC.

PAL comments (beginning with a semi–colon) should be omitted when you type in the programs in order to conserve disk and memory space. The BASIC REM statements at the beginning of the source files should be left in, however, as they contain documentation that you will use on a continuing basis. Another point to note is that the line numbers of the source programs are notably discontinuous. It is very important that you follow the numbering exactly when you type the programs in.

Our main task here is to set up the TransBASIC system for future use (though we'll also be getting a decent handful of new commands). This means typing in three files that you'll be using again and again: 'TRANSBASIC', 'TB/KERNEL', and 'TB/ADD.OBJ'. The first of these, 'TRANSBASIC', appears following this article as Program 4. It should be typed in as given, including the peculiar looking line 110, which is actually the token for the TransBASIC EXIT statement. How you type in the other two programs depends on whether you have handy a merge routine, like the one in Brad Templeton's POWER/MOREPOWER package, or the one by Glen Pearce that appeared in The Transactor last year (and on which the TransBASIC ADD statement is based). If you do have a merge routine, follow the instructions in Box 1. If you don't, follow the instructions in Box 2. Either way, when you're done you'll have in your disk directory the following new files:

```
 4     "TB/ADD.OBJ"      PRG
 23    "TB/ADD.SRC"      PRG
 15    "TB/KERNEL"       PRG
 10    "ADD"             PRG
```

All that remains is to type in the 'SCREEN THINGS' module, which is given as Program 3. You are now in a position to create your own TransBASIC dialect.

Box 1

Follow the instructions in this box if you have a merge routine available. The steps to be followed are:

1. Type in and save 'TRANSBASIC' (Program 4) if you have not already done so.

2. Type in and save 'TB/KERNEL' (Program 1).

3. Type in and save the module 'ADD' (Program 2).

4. Merge 'TB/KERNEL' with 'ADD' and alter line 95 of the combined result to read as follows:

```
95 XTRA .BYTE 3,0   ; STMTS,FNCS
```

Now save the combined program as 'TB/ADD.SRC'.

5. Assemble 'TB/ADD.SRC' with PAL, and save the object code as 'TB/ADD.OBJ'. This is the object file for the TransBASIC mini-dialect that enables the ADD command.

Box 2

Follow the instructions in this box if you don't have a merge routine. The steps to be followed are:

1. Type in and save 'TRANSBASIC' (Program 4) if you have not already done so.

2. Type in and save 'TB/KERNEL' (Program 1).

3. Without NEWing after step 2, type in the module 'ADD' (Program 3). Change line 95 of the combined result to read as follows:

```
95 XTRA .BYTE 3,0   ; STMTS,FNCS
```

Now save the combined program as 'TB/ADD.SRC'.

5. Assemble 'TB/ADD.SRC' with PAL, and save the object code as 'TB/ADD.OBJ'. This is the object file for the TransBASIC mini-dialect that enables the ADD command.

6. Delete from 'TB/ADD.SRC' all the lines belonging to 'TB/KERNEL'. Save the result of this operation as 'ADD'.

Using TransBASIC

This is the easy part. The first step is to load and run the program 'TransBASIC' (program 4). Now select the modules you need from those you have on disk (at present, of course, there are only two). Then, for each module, follow these steps:

1) Use the ADD statement to merge the module into memory, for example:

```
ADD "SCREEN THINGS"
```

(The only other alternative — until next issue — is: ADD "ADD").

2) List line 2 of your program. This line number is common to all modules. It will read something like:

```
REM 5 STATEMENTS, 0 FUNCTIONS
```

3) List line 95. This kernel line records the number of statements and functions in the TransBASIC that you are creating. When you first load in the kernel, line 95 reads:

```
95 XTRA .BYTE 2,0   ; STMTS,FNCS
```

(this is one line where I make an exception and retain the comment), indicating that the kernel contains two statements (ELSE and EXIT) and no functions. You are responsible for updating the two numbers appropriately as you add modules. After adding SCREEN THINGS, for instance, the first number in line 95 would be increased by five, the second would be left unchanged.

When you have finished adding modules, it would probably be a good idea to save the completed source file, at least temporarily. Then load PAL, if you haven't previously, and run your program. Normally the object code is origined to that popular niche at $C000, between BASIC and the I/O registers, but you can select another starting point if you wish (see line 31 of the source code). Save the object code directly, perhaps with Supermon, or convert it into DATA statements that can be loaded in with whatever program or programs you intend shall make use of the added commands. For the latter purpose I always use the Datafier, which you can find elsewhere in this issue, since it packs the DATA statements more compactly and unpacks them more promptly than is otherwise possible.

With that, the work is done. To activate the new commands type SYS 49152. Presto! — you have just extended BASIC to your own specifications, and now it's ready for use. Not only that but tomorrow you can do it again, with a different set of added commands, and again the day after that, to suit your momentary needs (at least, you will be able to when your collection of modules has grown a bit).

This kind of flexibility could have interesting consequences. Consider a set of TransBASIC modules encompassing many hundreds of commands. Is it likely that with that size of keyword vocabulary our notion of BASIC, and of programming languages in general, will remain unchanged? When we are able so readily to modify the language itself to an application, how much time proportionately will be spent on the actual coding, and how much on carefully considering which commands to enable?

More on those questions another time. Mundane matters demand our attention.

For one thing, a word of warning.

As I mentioned above, TransBASIC hijacks four important BASIC vectors and reroutes them through its own program space. Be-

ware, therefore, of overwriting your TransBASIC dialect with another program, or even with another TransBASIC dialect, as the vectors will be scrambled, and your computer will crash most agonizingly while you look helplessly on. The moral: always use the TransBASIC EXIT command to restore the old vectors before you do anything new with the memory where TransBASIC is roosting.

And now a look into the future.

From now on, new TransBASIC modules, each adding one or more commands to your collection, will be a regular feature in The Transactor. Some modules will deal with sprites, some with the SID chip. Some will offer sophisticated string handling commands that might be wanted, for example, in programming adventure games. Some modules may offer commands not available in any computer language yet existing, and not yet thought of by anyone except — you.

You see, I don't want to write all these new commands myself. Part of what TransBASIC is all about is that it offers a means whereby almost anyone can make a contribution to the growth of BASIC, our common tongue, and the rest of us can benefit. Many readers, I am sure, already have the expertise to write a new command; others will require only a little information to get them started (and will find it in the next issue). But even those whose machine language programming skills are rudimentary can contribute to the extent of sending in their ideas for commands they would like to see, even if they can't provide the code.

Next issue we'll look at some routines in the BASIC ROM that are helpful or necessary for writing TransBASIC commands, examine a map of TransBASIC line range allocations, respond to questions real or imagined, and whatever else fits. See you then.

• • • • •

New Commands

This part of the TransBASIC column is devoted to describing the new commands that will be added each issue. The descriptions follow a standard format:

The first line gives the command keyword, the type (statement or function), and a three digit serial number.

The second line gives the line range allotted to the execution routine for the command.

The third line gives the module in which the command is included.

The fourth line (and the following lines, if necessary) demonstrate the command syntax.

The remaining lines describe the command.

IF (Type: Statement Cat #: 000)
Line Range: 2474–2512
Module: TB/KERNEL
Example: IF A=3 THEN PRINT B
The IF statement is redefined to accept TransBASIC statements in the THEN–clause. Also, an ELSE has been added(001).

ELSE (Type: Statement Cat #: 001)
Line Range: 2514–2540
Module: TB/KERNEL
Example: IF A=3 THEN PRINT B: ELSE PRINT A
When the test–expression of an IF statement fails, the remainder of the line is scanned for a statement beginning with ELSE. If one is found, the statements following are executed. The first ELSE after the failed expression is the one used — ELSEs are not matched against IFs. Except after a failed IF statement, ELSE is referred to REM and has no effect.

EXIT (Type: Statement Cat #: 002)
Line Range: 2542–2558
Module: TB/KERNEL
Example: EXIT
Deactivates TransBASIC. TransBASIC may be reactivated with SYS 49152.

GROUND (Type: Statement Cat #: 013)
Line Range: 2740–2746
Module: SCREEN THINGS
Example: GROUND 2: REM RED BACKGROUND
The background colour is set to the specified colour.

FRAME (Type: Statement Cat #: 014)
Line Range: 2748–2754
Module: SCREEN THINGS
Example: FRAME 4: REM PURPLE BORDER
The border colour is set to the specified colour.

TEXT (Type: Statement Cat #: 015)
Line Range: 2756–2766
Module: SCREEN THINGS
Example: TEXT 1: REM WHITE PRINT
The screen print colour is set.

CRAM (Type: Statement Cat #: 016)
Line Range: 2768–2788
Module: SCREEN THINGS
Example: CRAM 0
Colour memory is filled with the specified value.

CLS (Type: Statement Cat #: 017)
Line Range: 2790–2832
Module: SCREEN THINGS
Example: CLS : REM CLEAR SCREEN, HOME CURSOR
Example: CLS 13 : REM CLEAR SCREEN, LINE 13 TO BOTTOM
Example: CLS 4,7: REM CLEAR LINES 4 THROUGH 7
In the second and third forms of the command the cursor position is not affected. All three forms clear colour memory to background colour in the range corresponding to the lines cleared on the screen.

ADD (Type: Statement Cat #: 055)
Line Range: 4474–4804
In Module(s): ADD
Example: ADD "VOCAB MANAGER"
Example: ADD "WITHIN",9
The named file is merged with the program in memory. If no device number is given the default is eight. The file number and secondary address used is $63 (99). The routine used is based on a program by Glen Pearce in The Transactor, volume 5, issue 2.

Program 1

```
  0 rem transbasic kernel (setp 27/84) :
  1 :
  2 rem 2 statements, 0 functions
  3 :
  4 rem keyword characters: 8
  5 :
  6 rem keyword    routine  line    ser #
  7 rem s/else     elsrtn   2514    001
  8 rem s/exit     ext      2542    002
  9 :
 10 rem utility: cifchr (2560/003)
 11 :
 12 rem kernel also includes modified
 13 rem 'if' statement (ifrtn/2474/000)
 14 :
 25 if peek(773)<192 goto 29: rem test if
      transbasic already enabled
 26 :
 27 ←a: rem 'exit' statement token
 28 :
 29 sys 700
 30 .opt oo
 31 *       =    $c000
 32 ;
 33 t2      =    2          ;transbasic takes over the
 34 t3      =    3          ;unused location at 2, and
 35 t4      =    4          ;the numeric conversion
 36 t5      =    5          ;vectors at 3 and 5 for its
 37 t6      =    6          ;zero page workspace
 38 ;
 91 ;
 92         jmp  start
 93         jmp  ext        ;emergency exit sys 49155
 94 ;
 95 xtra    .byte 2,0       ;stmts,funcs
 96 ;
 97         .byte 0
 98 ;
 99 skw     =    *          ;statement keyword list
100         .asc "elsEexiT"
598 ;
599 fkw     =    *          ;function keyword list
1096 ;
1097        .byte          0;keyword list terminator
1098 ;
1099 cmds    =    *          ;statement vectors
1100        .word $a93a,ext-1
1598 ;
1599 fncs    =    *          ;function vectors
2099 ;
2100 start   ldx  #7         ;swap out old
2102 stt1    lda  $304,x     ;basic vectors,
2104        sta  tvec,x     ;swap in new ones
2106        lda  nuvecs,x
2108        sta  $304,x
2110        dex
2112        bpl  stt1
2114        lda  xtra       ;calculate total
2116        clc             ;new keywords
2118        adc  xtra+1
2120        sta  xtra+2
2122        lda  #$60       ;disable this
2124        sta  start      ;routine with rts
2126        rts
2128 ;
2130 tvec    .word $a57c     ;storage for
2132 lvec    .word $a71a     ;old vectors
2134 evec    .word $a7e4
2136 fvec    .word $ae86
2138 ;
2140 nuvecs.word tok,lis,exc,fun ;new vectors
2142 ;
2144 tok     ldx  #0         ;tokenize
2146        ldy  #0
2148 tk1     lda  $200,x     ;fetch a byte
2150        cmp  #$22       ;check for open quote
2152        beq  tk9
2154        jsr  cifchr     ;check for alphabetic
2156        bcc  tk4        ;skip if not
2158        sty  $71        ;save .x and .y
2160        stx  t3
2162        ldy  #0
2164        sty  $0b        ;set up counter
2166        dey             ;set up indices
2168        dex
2170 tk2     iny
2172        inx
2174 tk3     lda  $200,x     ;fetch untokenized byte
2176        sec
2178        sbc  skw,y      ;compare with keyword list
2180        beq  tk2        ;go back for another byte
2182        cmp  #$80       ;check for match except high bit
2184        bne  tk5        ;try next keyword
2186        lda  #$5f       ;first byte of token,
2188        ldy  $71        ;the left arrow
2190        sta  $200,y
2192        iny
2194        lda  $0b        ;second byte of token
2196        adc  #$3f
2198 tk4     inx
2200        sta  $200,y
2202        iny
2204        lda  $200,y     ;loop unless line exhausted
2206        bne  tk1
2208        jmp  (tvec)     ;rejoin basic
2210 ;
2212 tk5     ldx  t3         ;find next keyword
2214 tk6     inc  $0b
2216        lda  $0b
2218        cmp  #$1e       ;skip past exponentiate token
2220        beq  tk6
2222 tk7     iny
2224        lda  skw-1,y
2226        bpl  tk7
2228        lda  skw,y
2230        bne  tk3
2232        ldy  $71        ;no matching keyword,
2234        lda  $200,x     ;give up attempt to
2236        bpl  tk4        ;tokenize
2238 ;
2240 tk8     lda  $200,x     ;scan for end of
2242        beq  tk4        ;line or end of
2244        cmp  #$22       ;quotes without
2246        beq  tk4        ;tokenizing
2248 tk9     sta  $200,y
2250        iny
2252        inx
2254        bne  tk8
2256 ;
2258 lis     cmp  #$5f       ;list — expand tokens
2260        bne  l2         ;skip if not left-arrow
2262        iny             ;get next byte,
2264        lda  ($5f),y    ;convert to token
2266        sbc  #$40       ;number
2268        cmp  #$1e
2270        bcc  l1
2272        sbc  #1
2274 l1      cmp  xtra+2     ;check if in bounds
2276        bcc  l3         ;go scan keyword list
2278        dey
2280        lda  ($5f),y    ;rejoin basic
2282 l2      tax
2284        jmp  (lvec)
2286 ;
2288 l3      sty  t3         ;countdown in .x
2290        ldy  #0         ;while scanning
2292        tax             ;keyword list
2294        beq  l5
2296 l4      lda  skw,y
2298        php
2300        iny
2302        plp
2304        bpl  l4         ;last keyword
2306        dex             ;character has bit
2308        bne  l4         ;7 set
2310 ;
2312 l5      lda  skw,y      ;print keyword
2314        php             ;using basic's
2316        and  #$7f       ;character-print
2318        jsr  $ab47      ;routine at $ab47
2320        iny
2322        plp
2324        bpl  l5
2326        ldy  t3
2328 l6      jmp  $a700      ;rejoin basic
2330 ;
2332 exc     jsr  $73        ;execute statement
2334 ex1     cmp  #$8b       ;'if' — handle at ifrtn
2336        beq  ex8
2338        cmp  #$5f       ;the left arrow
2340        bne  ex5        ;skip to rejoin basic
2342        jsr  $73        ;get next byte
2344        jsr  ex2        ;execute
2346        jmp  $a7ae      ;set up next statement
2348 ;
2350 ex2     sec             ;convert token to
2352        sbc  #$40       ;keyword number
2354        cmp  #$1e
2356        bcc  ex3
2358        sbc  #1
2360 ex3     cmp  xtra       ;check if in bounds
2362        bcs  ex7        ;syntax error
2364        asl
2366        tay
2368        lda  cmds+1,y   ;fetch vector
2370        pha             ;address from table
2372        lda  cmds,y
2374        pha             ;jump through chrget
2376        jmp  $73        ;to statement routine
2378 ;
2380 ex5     lda  $7a        ;decrement chrget
2382        bne  ex6        ;pointer
2384        dec  $7b
2386 ex6     dec  $7a
2388        jmp  (evec)     ;rejoin basic
2390 ;
2392 ex7     jmp  $af08      ;syntax error
2394 ;
2396 ex8     jsr  ex9        ;execute 'if'
2398        jmp  $a7ae      ;set up next statement
2400 ;
2402 ex9     lda  #>ifrtn-1  ;jump to
2404        pha             ;'if' execution
2406        lda  #<ifrtn-1  ;routine
2408        pha
2410        jmp  $73
2412 ;
2414 fun     jsr  $73        ;evaluate function
2416        cmp  #$5f       ;left arrow
2418        beq  fu2        ;evaluate transbasic funtion
2420        lda  $7a        ;decrement chrget
2422        bne  fu1        ;pointer,
2424        dec  $7b        ;rejoin basic
2426 fu1     dec  $7a
2428        jmp  (fvec)
2430 ;
2432 fu2     lda  #0         ;clear data type register
2434        sta  $0d
2436        jsr  $73        ;fetch keyword i.d. byte
2438        sec             ;convert to keyword number
2440        sbc  #$40
2442        cmp  #$1e
2444        bcc  fu3
2446        sbc  #1
2448 fu3     sec             ;check if in range
2450        sbc  xtra
2452        bcc  ex7        ;syntax error
2454        cmp  xtra+1
2456        bcs  ex7        ;syntax error
2458        asl             ;fetch function
2460        tay             ;vector from table
2462        lda  fncs+1,y
2464        pha
2466        lda  fncs,y
2468        pha             ;jump to function
2470        jmp  $73        ;routine through chrget
2472 ;
2474 ifrtn   jsr  $ad9e      ;evaluate test expression
2476        jsr  $79
2478        cmp  #$89       ;must be followed by
2480        beq  if1        ;goto ($89)
2482        lda  #$a7       ;or then ($a7)
2484        jsr  $aeff
2486 if1     jsr  $79        ;clear carry on numeric
2488        ldx  $61        ;check if test failed
2490        beq  elsrtn     ;yes — skip to else
2492        bcs  if2
2494        jmp  $a8a0      ;goto
2496 if2     pla             ;execute statement
2498        pla             ;after 'then'
2500        jsr  $79
2502        jmp  ex1
2512 ;
2514 elsrtn  jsr  $a8f8      ;skip statement
2516        jsr  $79
2518        cmp  #0         ;rts if end
2520        bne  els1       ;of line
2522        rts
2524 els1    jsr  $73        ;check for else
2526        cmp  #$5f       ;token
2528        bne  elsrtn
2530        jsr  $73
2532        cmp  #$40
2534        bne  elsrtn
2536        jsr  $73        ;execute
2538        jmp  if2
2540 ;
2542 ext     ldx  #7         ;exit routine
2544 ext1    lda  tvec,x     ;restore old
2546        sta  $304,x     ;vectors
2548        dex
2550        bpl  ext1
2552        lda  #$a2       ;restore ldx code to
2554        sta  start      ;enable start routine
2556        jmp  pdown      ;do powdown command if present
2558 ;
2560 cifchr  cmp  #$5b       ;return carry set
2562        bcc  cic1       ;if accumulator
2564        clc             ;contains
2566        bcc  cic2       ;alphabetic
2568 cic1    cmp  #$41
2570 cic2    rts
2572 ;
5222 pdown  rts             ;for users of 'power'
5224 ;
```

Program 2

```
  0 rem add (sept 27/84)          :
  1 :
  2 rem  1 statement, 0 functions
  3 :
  4 rem keyword characters: 3
  5 :
  6 rem keyword    routine  line  ser #
  7 rem add        xadd     4474055
  8 :
  9 rem e/iortns (39/056)
 10 :
 11 rem = = = = = = = = = = = = = = = = = =
 12 :
 39 setlfs  =   $ffba       ;i/o routine
 40 setnam  =   $ffbd       ;addresses in
 41 open    =   $e1c1       ;kernal rom
 42 chkin   =   $e11e
 43 close   =   $e1cc
 44 clrchn  =   $ffcc
 45 getin   =   $e124
 46 ;
 114        .asc "adD"
1114        .word xadd-1
4474 xadd   jsr  $ad9e      ;get filename
4476        jsr  $b6a3
4478        sta  t3         ;save length
4480        txa
4482        pha
4484        tya
4486        pha
4488        lda  t3
4490        jsr  $b47d      ;reserve space
4492        tax
4494        bne  xa1
4496        jmp  $af08      ;null string syntax error
4498 xa1    clc             ;add ,p to filename
4500        adc  #2
4502        sta  t3
4504        jsr  $b47d
4506        tay
4508        pla
4510        sta  $23
4512        pla
4514        sta  $22
4516        dey
4518        lda  #"p"
4520        sta  ($62),y
4522        dey
4524        lda  #","
4526        sta  ($62),y
4528 xa2    dey
4530        bmi  xa3
4532        lda  ($22),y
4534        sta  ($62),y
4536        bne  xa2
4538 xa3    lda  t3         ;set up for setnam
4540        ldx  $62
4542        ldy  $63
4544        jsr  setnam
4546        jsr  $79        ;check for
4548        beq  xa4        ;device number
4550        jsr  $aefd
4552        jsr  $b79e
4554        .byte $2c
4556 xa4    ldx  #8         ;default device 8
4558        lda  #$63       ;file number and
4560        tay             ;secondary addr 99
4562        jsr  setlfs
4564        jsr  open
4566        ldx  #$63
4568        jsr  chkin      ;open channel
4570        jsr  dskget     ;skip load address
4572        jsr  dskget
4574 xa5    jsr  dskget     ;line link low
4578        jsr  dskget     ;end on zero high byte
4580        bne  xa7
4582        jsr  clsclr     ;wrap up i/o
4584        jsr  $a659      ;reset chrget ptr, clr
4586        jmp  $e386      ;'ready.'
4590 xa7    jsr  dskget     ;save line number
4592        sta  $14
4594        jsr  dskget
4596        sta  $15
4598        ldy  #0
4600 xa8    jsr  dskget     ;move rest of
4602        sta  $200,y     ;line to input
4604        beq  xa9        ;buffer
4606        iny
4608        bne  xa8
4610 xa9    tya             ;save line size
4612        clc
4614        adc  #5
4616        sta  $0b
4618        jsr  $a613      ;search for line #
4620        bcc  xa13       ;skip if not found
4622        ldy  #1
4624        lda  ($5f),y    ;delete line
4626        sta  $23
4628        lda  $2d
4630        sta  $22
4632        lda  $60
4634        sta  $25
4636        lda  $5f
4638        dey
4640        sbc  ($5f),y
4642        clc
4644        adc  $2d
4646        sta  $2d
4648        sta  $24
4650        lda  $2e
4652        adc  #$ff
4654        sta  $2e
4656        sbc  $60
4658        tax
4660        sec
4662 xa10   bcc  xa5        ;link
4664        lda  $5f
4666        sbc  $2d
4668        tay
4670        bcs  xa11
4672        inx
4674        dec  $25
4676 xa11   clc
4678        adc  $22
4680        bcc  xa12
4682        dec  $23
4684        clc
4686 xa12   lda  ($22),y
4688        sta  ($24),y
4690        iny
4692        bne  xa12
4694        inc  $23
4696        inc  $25
4698        dex
4700        bne  xa12
4702 xa13   jsr  $a663      ;clr
4704        jsr  $a533      ;re-link
4706        clc
4708        lda  $2d
4710        sta  $5a
4712        adc  $0b
4714        sta  $58
4716        ldy  $2e
4718        sty  $5b
4720        bcc  xa14
4722        iny
4724 xa14   sty  $59        ;make space for
4726        jsr  $a3b8      ;new line
4728        lda  $14
4730        ldy  $15
4732        sta  $1fe
4734        sty  $1ff
4736        lda  $31
4738        ldy  $32
4740        sta  $2d
4742        sty  $2e
4744        ldy  $0b
4746        dey
4748 xa15   lda  $01fc,y    ;move line
4750        sta  ($5f),y    ;into program
4752        dey
4754        bpl  xa15
4756        jsr  $a663      ;clr
4758        jsr  $a533      ;re-link
4760        clc
4762        bcc  xa10       ;do another
4764 ;
4766 dskget jsr  getin      ;get byte
4768        pha
4770        lda  $90        ;check status
4772        and  #$bf
4774        bne  dkg1
4776        pla             ;return if ok
4778        rts
4780 dkg1   jsr  clsclr     ;wrap up i/o
4782        ldx  #$1d       ;'merge error'
4784        lda  #<mrgerr
4786        sta  $22
4788        lda  #>mrgerr
4790        jmp  $a445
4792 ;
4794 mrgerr .asc "mergE"
4796 ;
4798 clsclr  lda  #$63       ;close file
4800        jsr  close      ;clear channels
4802        jmp  clrchn
4804 ;
```

Program 3

```
  0 rem screen things (aug 25/84)    :
  1 :
  2 rem 5 statements, 0 functions
  3 :
  4 rem keyword characters: 22
  5 :
  6 rem keyword    routine line    ser #
  7 rem s/ground   grd     2740   013
  8 rem s/frame    fram    2748   014
  9 rem s/text     tex     2756   015
 10 rem s/cram     cfill   2768   016
 11 rem s/cls      clea    2790   017
 12 :
 13 rem = = = = = = = = = = = = = = = = = =
 14 :
 104        .asc "grounDframEtexT"
 105        .asc "craMclS"
1104        .word grd-1,fram-1,tex-1
1105        .word cfill-1,clea-1
2740 grd    jsr  $b79e      ;get byte in .x
2742        stx  $d021      ;put in background
2744        rts             ;colour register
2746 ;
2748 fram   jsr  $b79e      ;get byte in .x
2750        stx  $d020      ;put in border
2752        rts             ;colour register
2754 ;
2756 tex    jsr  $b79e      ;get byte in .x
2758        txa
2760        and  #$0f       ;put low byte in
2762        sta  $286       ;text colr register
2764        rts
2766 ;
2768 cfill  jsr  $b79e      ;get byte in .x
2770        txa
2772        ldy  #0
2774 cf1    sta  $d800,y    ;fill colour
2776        sta  $d900,y    ;memory
2778        sta  $da00,y
2780        sta  $db00,y
2782        iny
2784        bne  cf1
2786        rts
2788 ;
2790 clea   bne  cle1       ;if no parameters
2792        jmp  $e544      ;just clr screen
2794 cle1   jsr  $b79e      ;get a byte
2796        stx  $14        ;check range
2798        cpx  #$19
2800        bcs  cle5
2802        jsr  $79        ;branch if no
2804        beq  cle3       ;2nd parameter
2806        jsr  $aefd      ;check for comma
2808        jsr  $b79e      ;get 2nd parameter
2810 cle2   cpx  $14        ;exit if less
2812        bcc  cle4       ;than first one
2814        cpx  #$19       ;check range
2816        bcs  cle5
2818        .byte $2c
2820 cle3   ldx  #$18       ;default 2nd param
2822        jsr  $e9ff      ;clear a line
2824        dex
2826        bpl  cle2
2828 cle4   jmp  $e56c      ;restore cursor
2830 cle5   jmp  $b248      ;illegal quantity
2832 ;
```

Program 4

```
100 if peek(773)<192 goto 120
110 ←a        : rem  exit
120 if a=1 goto 140
130 a=1: load "tb/add.obj",8,1
140 sys 49152  : rem enable tb/add
150 a$=chr$(34)+"tb/kernel"+chr$(34)
160 print "[redacted]load " a$ ",8[redacted]"
170 poke 198,1
180 poke 631,13
```

Hardware Corner

Domenic DeFrancesco

Input and the Keyboard Matrix

In the previous Hardware Corner we covered output on the parallel port and connected 8 LEDs as an example. This article is a tutorial and no construction project will be presented. We will cover input from the parallel port and look into reading the computer's built-in keyboard by software.

Simple input: Reading Switches

Connecting a small number of switches (up to 8) is a fairly simple process. Each switch must be connected between an I/O line and ground. The switches used may be pushbuttons (as in the case of a keyboard), toggle switches, relay contacts, or any open/close circuit element. Actually connecting the switches to the I/O lines can be complicated by a few considerations:

A 'pull-up' resistor is sometimes required depending on the internal workings of the I/O chip being employed. The 8 peripheral I/O lines on the user port of all Commodore computers have this pull-up resistor built into the chip, and therefore, it is not required in our application. Pull-up resistors vary in value from 1k to 10k ohms.

A 'series damping' resistor, used to reduce transient voltages, is optional, but recommended when the switch is far from the computer, or when long-term reliabilty is important. Typical values for series damping resistors are between 27 ohms and 100 ohms. See figure 1 for a schematic showing the different resistor arrangements.

To read the state of the switches, simply set the data direction register (ddr) to input (zero), and PEEK the data register (dr). You may want to refresh your memory of the operation of the ddr and dr by referring to the first Hardware Corner in Vol. 5, Issue 01, or the second in Issue 03. The following BASIC subroutine, given a bit number 'BN' from zero through seven, will assign variable 'D' with a 1 or 0 depending on whether the specified switch is open or closed, respectively.

```
1000 rem** read switch bn (0-7) **
1010 d = peek(dr) and 2↑bn
1020 rem  read bit 'bn' of data reg.
1030 d = -(d>0)
1040 rem  make d either 1 or 0
1050 return
```

Here's an example of a program which uses the above subroutine, and reports on the state of eight switches, each connected to an I/O line on the parallel port.

```
10 rem** read 8 switches on user port
20 rem* choose appropriate line below
30 ddr = 59459: dr = 59471: rem pet/cbm
40 ddr = 37138: dr = 37136: rem vic
50 ddr = 56579: dr = 56577: rem c64
60 :
70 poke ddr,0: rem all lines inputs
80 for bn = 0 to 7: rem all 7 switches
90 gosub 1000   : rem read switch #bn
100 print " switch  " bn;
110 if d = 1 then print " : off (open) "
120 if d = 0 then print " : on  (closed) "
130 next bn
140 end
```

The Keyboard Matrix

The method outlined above is good for connecting a small number of switches, but can become very expensive if many switches (like a keyboard), need to be connected, since several I/O chips will be required (along with the necessary extra space on the circuit board).

The alternative method is to connect the switches in a matrix arrangement. Figure 2 shows 16 pushbutton switches connected to the user port while using only 8 lines. With the addition of one more line (shown in dashes), 20 pushbuttons can be connected. This set-up uses far fewer I/O lines, requiring less chips, circuitry, and connections. The two possible disadvantages are: 1) more software is required to read the state of the pushbutton, and 2) the software may not be able to determine which pushbuttons are closed if 3 or more keys are pressed.

Reading the 16 keys in figure 2 is a little bit more complicated than reading the direct-connected switches as above. The general idea behind "scanning" a matrix is that there are two sets of lines: The row select lines, and the keyboard column input lines. The row select lines are outputs, and determine which row of keys will be read on the input lines. The input lines are, obviously, set as inputs, and are used to determine which key in the desired row is depressed.

The scanning program generally uses two nested loops: the main, outside loop iterates once for each row in the matrix, and the inner loop is performed as many times as there are keys in a row.

As you can see in figure 2, PB4-PB7 are used as the row select lines, and PB0 to PB3 are the column input lines. The data

direction register must be set up accordingly: 11110000 (binary). The scanning program must select each row in turn by bringing the appropriate row select line LOW, and then testing each column input line to see if it has gone LOW. If it has, the corresponding key was pressed. So, we could start by making PB4 low and PB5–PB7 high. A zero bit in the input lines will indicate a pressed switch, and the bit's position will indicate which one. But a program is worth a thousand words, so before this gets confusing, I'll let you see for yourself how it's done.

The program listed below scans the keyboard matrix shown in Figure 2, and assigns the number of a key pressed (from 0 to 15) to the variable 'D'. Depending on which machine the keyboard is connected to, you must set up the variables 'DR' and 'DDR' accordingly.

```
100 rem* read 4 by 4 keyboard matrix *
110 poke ddr,240: rem* data direction
120 d = 15 :rem * key pressed
130 for row = 0 to 3
140 poke dr,16*(15-2↑row) + 15
150 keys = peek(dr) and 15
160 if keys = 15 goto 200
170 for col = 0 to 3
180 if(keys and 2↑col) = 0 then
        d = col + 4*row
190 next col
200 rem— endif —
210 next row
220 end
```

The Commodore 64's Keyboard

All the Commodore computers use a matrix arrangement for their keyboards. On the C64, the keyboard row is selected with the register at 56320, and the keyboard input lines are read from location 56321. Note that these ports are the same ones used for the joysticks, which explains why you can't properly type on the keyboard while playing around with the joystick.

The 64's keyboard is arranged as an 8 by 8 matrix, giving a total of 64 keys. To select a keyboard row, the corresponding bit in the row select register must be set to zero. The state of that keyboard row can then be read from location 56321. The keys corresponding to the given rows and columns are shown in the following table. The row number appears in the leftmost column, followed by the value that would be stored in the select register 56320 (255–2↑(row number)).

D600-select

ROW	Column (bit in location 56321)							
	0	1	2	3	4	5	6	7
0 (254)	DEL	rtrn	rt	F7	F1	F3	F5	dn
1 (253)	3	W	A	4	Z	S	E	l. shft
2 (251)	5	R	D	6	C	F	T	X
3 (247)	7	Y	G	8	B	H	U	V
4 (239)	9	I	J	0	M	K	O	N
5 (223)	+	P	L	-	.	:	@	,
6 (191)	\	*	;	HOME	r.shf	=	↑	/
7 (127)	1	←	CTRL	2	SPACE	C=	Q	STOP

Notes:
1) The shift lock key is connected with the left shift key.
2) The RESTORE key is not part of the keyboard matrix, but is directly wired to generate an NMI (Non Maskable Interrupt) when struck.

Scanning the built–in keyboard is no more difficult than scanning our custom–made 16 key example. It might be sometimes desirable to scan the keyboard yourself instead of relying on the built–in routine. For example, you may wish to check for only a specific key, or check the current key pressed without affecting the keyboard buffer. Also, a custom keyboard–scanning routine will allow you to detect two keys which are simultaneously depressed; a capability that is often desired in games.

To show how easy it is to scan the matrix, the scanning program is presented in BASIC. When scanning the keyboard, either from machine language or BASIC, it is necessary to disable the IRQs to prevent the system's keyboard scanning routine from interfering. This is done in BASIC by disabling the timer which generates the interrupts. The scanning program below will scan the keyboard and return a code (the same as from PEEK(197)) for the key pressed in the array variable K(0). If two keys are pressed, another value will be stored in K(1). The number of keys pressed, 0, 1 or 2, will be stored in the variable 'C'. The keyboard scan is done by the subroutine starting at line 1000.

```
100 rem* scan c64 keyboard
110 rem*
120 sel = 56320:rem* kbd row select reg
130 inp = 56321:rem* kbd input register
140 for i = 0to7: e(i) = 2↑i: next i
150 rem* e() is exponent array (2↑x)
170 :
180 print " # keys " , " 1st key " , " 2nd key "
181 :
190 for loop = 0 to 1 step 0
200 gosub370: rem* read keyboard
210 print c, k(0), k(1)
220 next loop
230 end
250 :
260 rem*********************************
270 rem*    keyboard scan subroutine    *
280 rem*     set up e() before calling.  *
290 rem*          input variables:       *
300 rem*    " SEL " : row select register *
310 rem*    " INP " : keyboard input reg  *
320 rem*          output variables:      *
330 rem*    " C " : # of keys down (0–2)  *
340 rem*    " K(0) " : 1st key down or 64 *
350 rem*    " K(1) " : 2nd key down or 64 *
360 rem*********************************
370 rem– this is the entry point –
380 :
390 poke 56334,peek(56334)and254
400 rem* turn interrupts off
410 :
420 c = 0: rem* # keys pressed (0–2)
430 k(0) = 64: k(1) = 64:rem* 64 = no key
440 poke sel,0
450 rem* test for no keys pressed
460 if peek(inp) = 255 then return
470 :
480 for row = 0 to 7
490 poke sel,255–e(row)
500 keys = peek(inp):if keys = 255goto550
510 for col = 0 to 7
520 if(keys and e(col)) = 0 and c<2 then
        k(c) = row*8 + col: c = c + 1
530 :
540 next col
550 next row: rem—endif—
560 :
570 poke56334,peek(56334)or1
580 rem* turn interrupts back on
590 return
```

The C64's I/O ports

The scanning procedure is pretty straightforward, but you may have noticed something a little wierd. Location 56320 is used as an output, since we POKE it with a value to select a keyboard row. But we can also PEEK the very same location, and it will give the current status of the joystick plugged into port #2. All this without ever touching a data-direction register. What's going on here? Well, the magic is due to the way the 6526 I/O chip works. If a line is set as an output (through the data direction register) and is set high via the data register, you would expect a PEEK of the location to show that line as being high, no matter what. That's not necessarily the case with the 6526: when the port is read with a PEEK, the value obtained is not the contents of the I/O register, it is the actual state of the I/O pins. Thus, if a line is being pulled low (in the case of the C64, by the joystick), even if it is an output and set high it will read as a zero. This is how a line can be used for input or output without changing the data direction register.

Port A of the 6526 has what's known as "passive pull-up" I/O lines. Such lines have an internal pull-up resistor, which pulls the line high when set as an input or when set as an output and high. A passive pull-up line can be pulled low externally without damaging the I/O chip, even if it's supposed to be an output. That's why it's not as awful as it sounds to use the joystick to force output lines low.

While we're on the subject of I/O line pull-up, there are also those of the "active pull-up" variety (as you might have guessed). An active pull-up line is pulled high by an active circuit element like a transistor when it's set as an output and high. Active pull-up lines can't be pulled low externally like passive pull-up lines can. Not unless you want to blow your I/O chip! Some chips have I/O lines with both active and passive pull-ups. These lines give the best of both worlds, since they can sink a lot of current when in the high state as outputs, and they will "float" high when set as inputs.

The PET/CBM keyboard matrix

The keyboard matrix on PET/CBM machines has 10 rows by 8 columns, and works in basically the same way as the 64's. The main difference is that the row select register uses a *decoder* chip to save I/O lines. The decoder will bring a given line low, given the binary value of that line on it's inputs. For us software folks on the outside of the machine, that means that we just store a number from 0 to 9 in the row select register to select one of ten lines. We don't have to calculate the proper value to affect a certain line.

Specifically, the row select register on PETs is found in the lower 4 bits of location 59408, and the column inputs are read from location 59410. The scanning process for this keyboard is identical to that for the 64's except for the POKE which selects the row. The advantage to the PET's layout is that it uses fewer I/O lines. The advantage to the 64's scheme is that a quick check can be made to see if ANY key is pressed, without checking each row in turn (storing a zero in the row select register allows you to read the state of ALL keys at once). So the PET's keyboard takes longer to scan when no keys are being pressed.

Wrapping up

The topic of input in general could cover many more pages, but the above was just a collection of a few simple and advanced concepts. Input to the computer is not usually discussed on the lowest hardware level, but serious computerists should know what's going on in there. Reading devices such as the keyboard directly with your own software gives you just a little bit more control over your machine. Power! The next Hardware Corner (which will hopefully appear in the NEXT issue this time) will present a couple of construction projects: An RS-232 interface, and direct connection of a parallel printer to the user port.

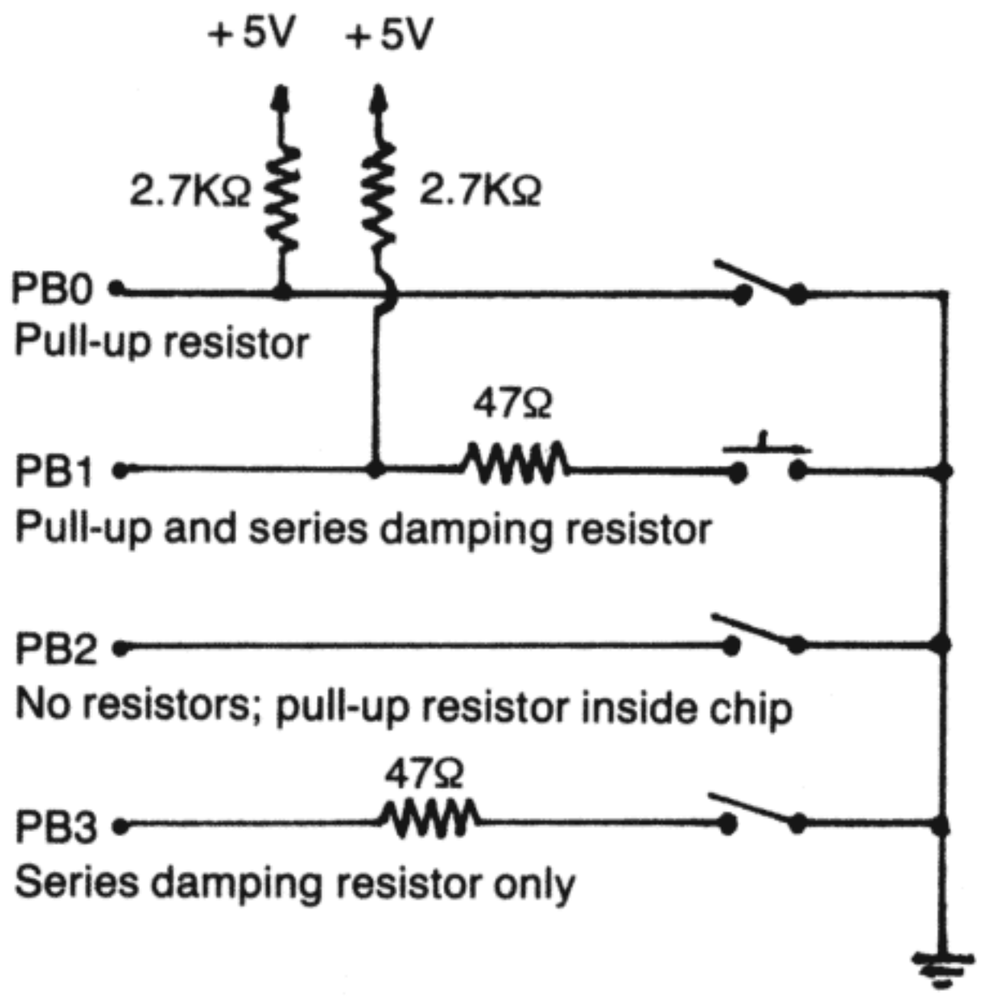


Figure 1:
Pull-up and series damping resistor connections

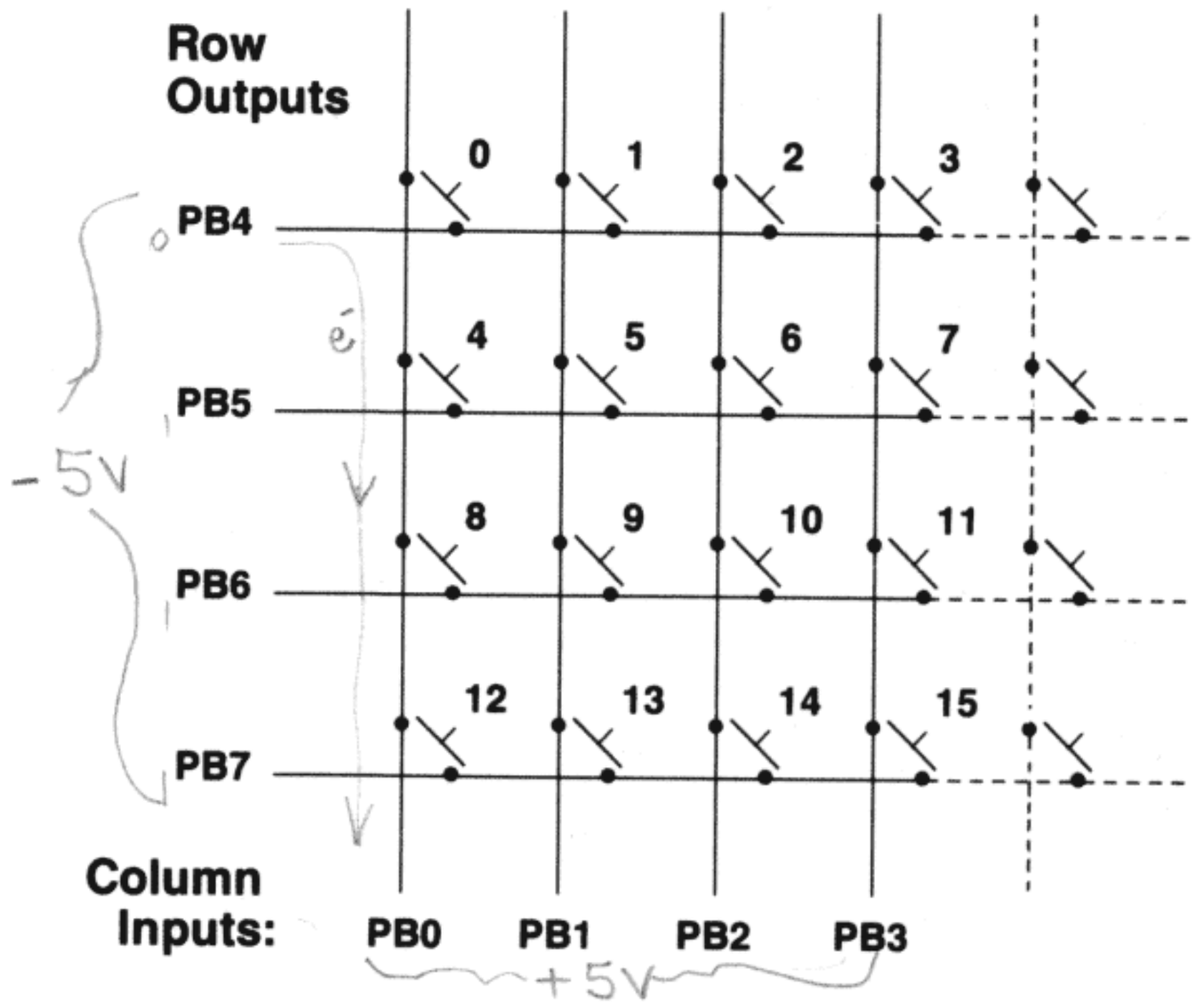


Figure 2: A 4 by 4 keyboard matrix.
Note how one additional I/O line allows 4 more keys.

The Commodore 64 Keyboard Part 1: The KERNEL Routines

Aubrey Stanley
Mississauga, Ontario

Its amazing how something as ordinary as a keyboard can turn you off what must be the greatest PC for its price on the market today. I work in software development, which means that I use the keyboard a lot. I'm used to keyboards that respond to keys as they are entered, no matter how many previous keys are pressed. This is known as n–KEY ROLLOVER in computer jargon. I found the one–key–at–a–time approach very frustrating. To compound the situation, my five–year–old daughter, Charmaine, whom I was teaching to type, decided to end her relationship with the Commodore because "it couldn't spell".

While investigating her problems I soon discovered that all her complaints originated in this one–key–at–a–time methodology that Commodore uses to drive the 64 keyboard. It was truly a lesson to see her refuse to accept a situation which I, as a conditioned adult, had become resigned to. Yes, the computer cannot spell! If two keys are down at the same time, chances are even that the second key pressed will not display until the first key is released. This can slow an average typist down by 50% or more. In the case where the second key does display, if the second key is released before the first key, then the first key will be displayed again. And if, perchance, there are three keys down simultaneously (you're a super typist or a child), then it's likely that the third key will display incorrectly – try 'GHJ' for instance.

I decided I had to do something to restore my child's faith in computers. The PROGRAMMERS REFERENCE GUIDE had precious little to say on the keyboard. Even the good old TRANSACTOR mags were decidedly mute on the subject. So armed with a newly purchased Monitor, I took a plunge into the murky waters of the Kernel ROM. My investigations form the basis of this first article on the Commodore 64 keyboard. Illustrated here is the software interface, the associated Kernel routines and some interesting facts that have been left out of official Commodore publications. The second article will build upon this knowledge by developing an alternative software driver with n–Key Rollover capability and the ability to easily redefine the standard arrangement of the keyboard. Article three will demonstrate further routines that run as extensions of this keyboard driver and enhance the capabilities of the standard Screen Editor. You will be able to define any key to generate an 'instant keyword' of your choice – e.g. type shifted 'G' and get 'GOSUB'. Or erase to the end–of–line. Or even link in your own routine to perform an editing task you sorely miss in the standard editor.

How Keys Are Read

Imagine each key as a switch which turns an individual bit in memory to '0' when pressed. If the key is released, the memory bit reverts to '1'. Now since there are 64 keys (RESTORE excepted), the keyboard may be thought of as occupying 64 bits, or 8 bytes, of memory, each bit of which represents a particular key. It's the physical position of the key that I am talking about, not the symbol etched on it. It's important to understand this. Any one byte of this pseudo memory is available to the software at location $DC01. Which byte the software reads is decided by writing a bit pattern to location $DC00 before performing the read*. The bit pattern consists of seven '1' bits and one '0' bit. Just where the '0' is positioned will determine the set of eight keys that will be read from $DC01. Table 1 illustrates this process using the designations of the standard keyboard. But again, remember that the letter on the key is only the symbol, its the physical position of the key that determines which particular bit is set or reset in $DC01.

** Editor's Note: $DC00 and $DC01 are hardware registers in one of the I/O chips, and control the I/O ports to which the keyboard switches are connected. See 'Hardware Corner' in this issue for more on this.*

Table 1: Keyboard Switch Matrix

ROW	Column (bit in location 56321)							
	7	6	5	4	3	2	1	0
$FE	dn	F5	F3	F1	F7	rt	rtrn	DEL
$FD	l. shft	E	S	Z	4	A	W	3
$FB	X	T	F	C	6	D	R	5
$F7	V	U	H	B	8	G	Y	7
$EF	N	O	K	M	0	J	I	9
$DF	,	@	:	.	-	L	P	+
$BF	/	↑	=	r.shf	HOME	;	*	\
$7F	STOP	Q	C=	SPACE	2	CTRL	←	1

Index Of A Key

The Kernel determines the index of a key by effectively adding its column position to its row position multiplied by 8. Thus the key 'B' will have an index of 4+(3*8)=28 or $1C. The index is used to access a table from which is fetched the ASCII or CHR$ value of the key. For the 'B' key this would normally be 66 or $42. Certain keys are given special CHR$ values in the table. These are '1' for a Shift Key, '2' for the Commodore Key and '4' for the Control Key.

REPLACING THE KERNEL'S DECODE TABLES IS ONE WAY OF RE–CONFIGURING THE KEYBOARD.

Interrupt Entry

The Kernel, when it initializes the system, sets up one of the timers to interrupt at 1/60th second intervals. Each time this interrupt occurs, the hardware automatically saves the Program Counter of the current task together with the status register, and jumps to the vector stored in $FFFE–$FFFF. This vector points to $FF48 where the Kernel saves the registers and then does an indirect jump through location $314–$315. This vector points to $EA31, the beginning of the routines that increment the real time clock, flash the cursor, handle the tape motor and finally, scan the keyboard.

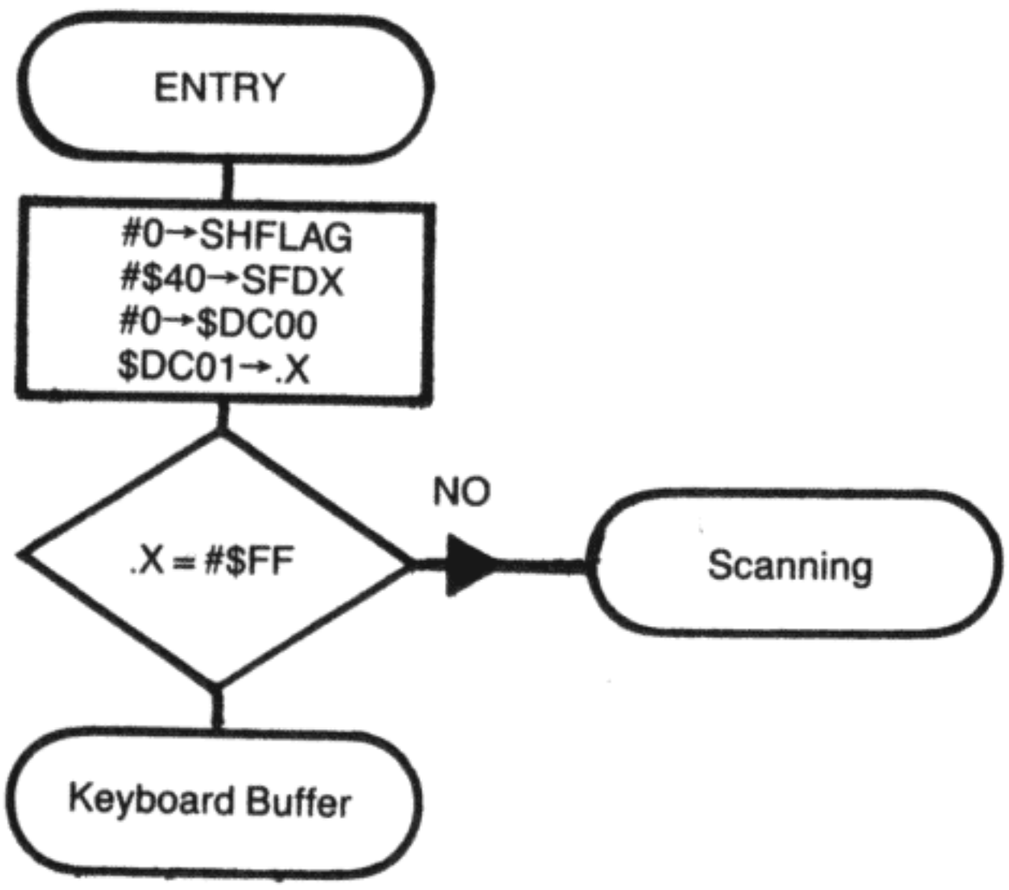
It's the keyboard scanning routine that we're concerned with here. A detailed explanation of this routine follows.

KERNEL Keyboard Variables

The Kernel uses low memory to store its variables. The use of each of these variables will become clear in the descriptions which follow.

```
LSTX    = $C5   ;Index of Last Key Pressed
NDX     = $C6   ;Number of Characters in Keyboard Buffer
SFDX    = $CB   ;Index of Current Key Pressed
KEYTAB  = $F5   ;Keyboard Decode Table Pointer
KEYD    = $277  ;Keyboard Buffer Start
XMAX    = $289  ;Max. Characters Allowed in Keyboard Buffer
RPTFLG  = $28A  ;Flag to Enable/Disable Repeats
KOUNT   = $28B  ;Repeat Speed Count
DELAY   = $28C  ;Delay to Repeat New Key Press
SHFLAG  = $28D  ;Current State of Shift, C= & Control Keys
LSTSHF  = $28E  ;Last State of Shift, C= and Control Keys
KEYLOG  = $28F  ;Vector for Keyboard Table Setup
MODE    = $291  ;Flag to Enable/Disable Character Set Switch
CINV    = $314  ;Vector for Hardware IRQ Interrupt
```

Keyboard Driver Entry

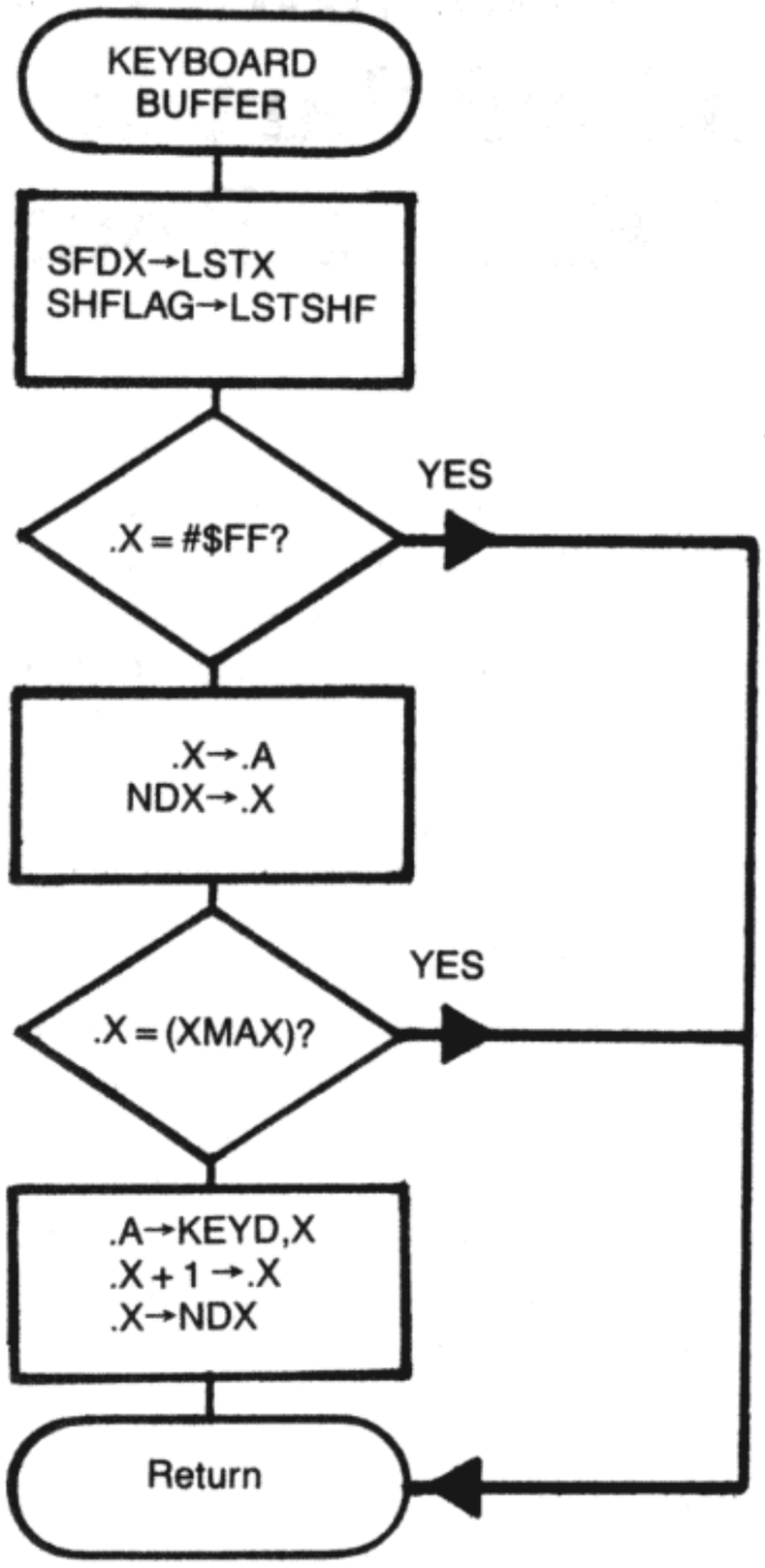


The Keyboard routines are entered at $EA87 via a subroutine call from location $EA7B. This is unfortunate, because the lack of a vector here means that anyone wishing to replace the standard driver must take over the entire interrupt routine, performing all the other housekeeping tasks as well.

First SHFLAG is cleared to 0. SHFLAG holds the run–time state of the Shift, Commodore and Control keys. A value of $40 is then written to SFDX. SFDX holds the current index of the key found to be pressed during the scanning operation. $40 (decimal 64) is the index when no keys are pressed. Next a value of $00 is written to location $DC00. Location $DC01 is then read. If this produces an $FF result, then it means that no keys are pressed. Any other value signifies that at least one key is pressed. If no keys are pressed, the driver enters the KEYBOARD BUFFER routine, otherwise the SCANNING routine is entered.

$00 IS A SPECIAL BIT PATTERN WRITTEN TO LOCATION $DC00 TO DETERMINE IF ANY (IT DOESN'T MATTER WHICH) KEYS ARE PRESSED.

The KEYBOARD BUFFER Routine



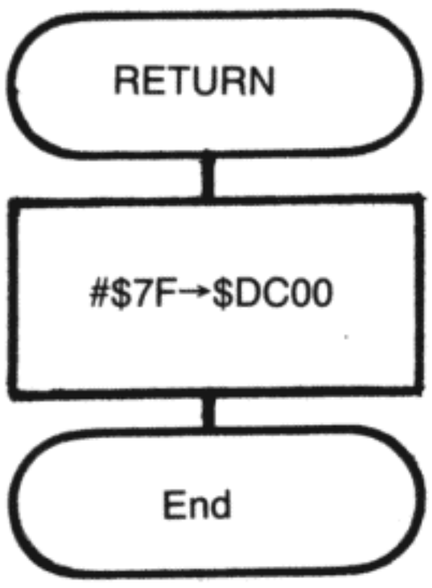
This routine is entered at $EB26. Here LSTSHF and LSTX are updated to reflect the new state of the keyboard. LSTSHF stores the states of the Shift, Commodore and Control keys (in bits 0, 1 and 2 respectively) from the last activation of the interrupt routine. If the bit is '0' then this means that the corresponding key is NOT pressed. LSTX holds the index of the key–press (excluding the Shift, Commodore and Control keys) from the last activation of the interrupt routine. Remember again that the index gives the physical position of the key. (according to its position in Table 1)

SHFLAG is copied to LSTSHF and SFDX to LSTX. At this point register 'X' will hold the CHR$ value (taking into account the state of the Shift, Commodore and Control keys) of the key that was found to be pressed during the SCANNING routine. If no key was pressed then 'X' will contain the value $FF. Nothing is put into the keyboard buffer if the value of 'X' is $FF or if NDX has reached XMAX, the latter condition indicating that the keyboard buffer is full. Otherwise the CHR$ value is inserted into KEYD, the keyboard buffer, at the location offset by NDX, and NDX is incremented. Finally, the KEYBOARD BUFFER routine exits to the RETURN routine.

'LSTSHF' AND 'LSTX' MAY BE EXAMINED TO SEE JUST WHICH KEYS ARE CURRENTLY PRESSED.

'KEYD' IS THE QUEUE THAT THE KERNEL USES TO FEED KEYS TO BASIC AND OTHER PROGRAMS.

The RETURN Routine



The RETURN routine is entered at $EB42. It writes $7F to location $DC00 and then does a Return From Subroutine which ends the Keyboard servicing portion of the interrupt routine. Return is to location $EA7E where the Kernel reads the CIA Interrupt Control Register (from $DC0D), restores the CPU registers and exits the interrupt with an RTI instruction.

The bit pattern of $7F written to location $DC00 allows BASIC or any other program to read directly (from $DC01), the set of eight keys which includes the CONTROL and STOP keys.

IT IS WORTH MENTIONING HERE THAT THE SCREEN SCROLLING ROUTINE RESETS $DC00 to $7F.

The SCANNING Routine

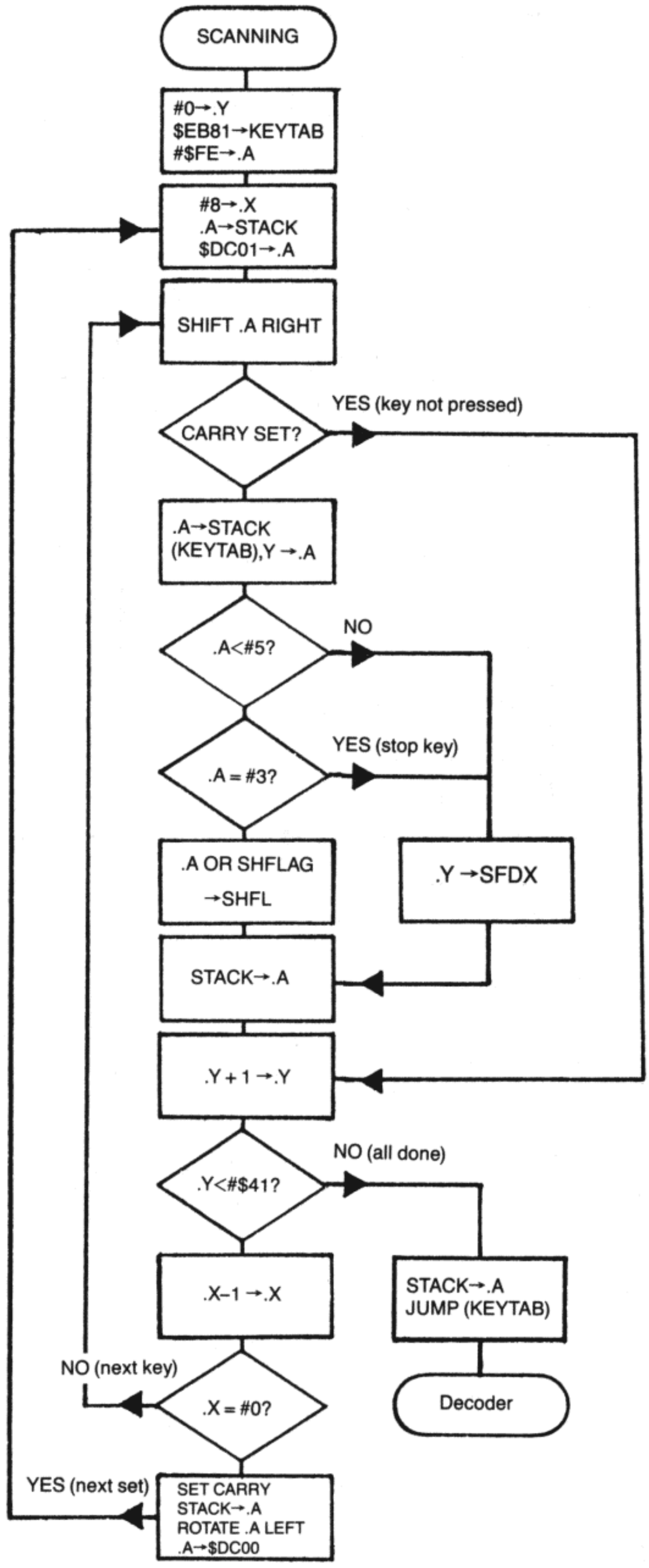
The SCANNING routine is entered at $EA9A. First KEYTAB is set up to point to the Decode Table at $EB81. This table gives the CHR$ values for normal (unshifted) keys. The keys are scanned in sequence beginning at index = '0' and ending at index = '63' ($3F). This is achieved in eight activations of an outer loop, each of which writes the appropriate bit pattern to location $DC00 and then enters an inner loop where the corresponding set of eight keys are read from location $DC01 and examined one by one.

In the inner loop, if the value of the key bit = '0' (i.e. the key is pressed), then its index is used to fetch the CHR$ value from the decode table whose pointer had been set up in KEYTAB. If the CHR$ value is '1', '2' or '4' (Shift, Commodore or Control), then this value is OR'd into SHFLAG which maintains the values of these keys. Otherwise the index is itself written to SFDX, overwriting any previous index that may have been stored there during the scanning process. THIS FACTOR EXPLAINS THE ONE-KEY-AT-A-TIME FEATURE OF THE COMMODORE KEYBOARD AND IS THE ROOT CAUSE OF ALL THE ANOMOLIES. After the scan is completed, SFDX will contain the index of the pressed key with the highest index, while SHFLAG will hold the states of the Shift, Commodore and Control keys in the least three significant bits.

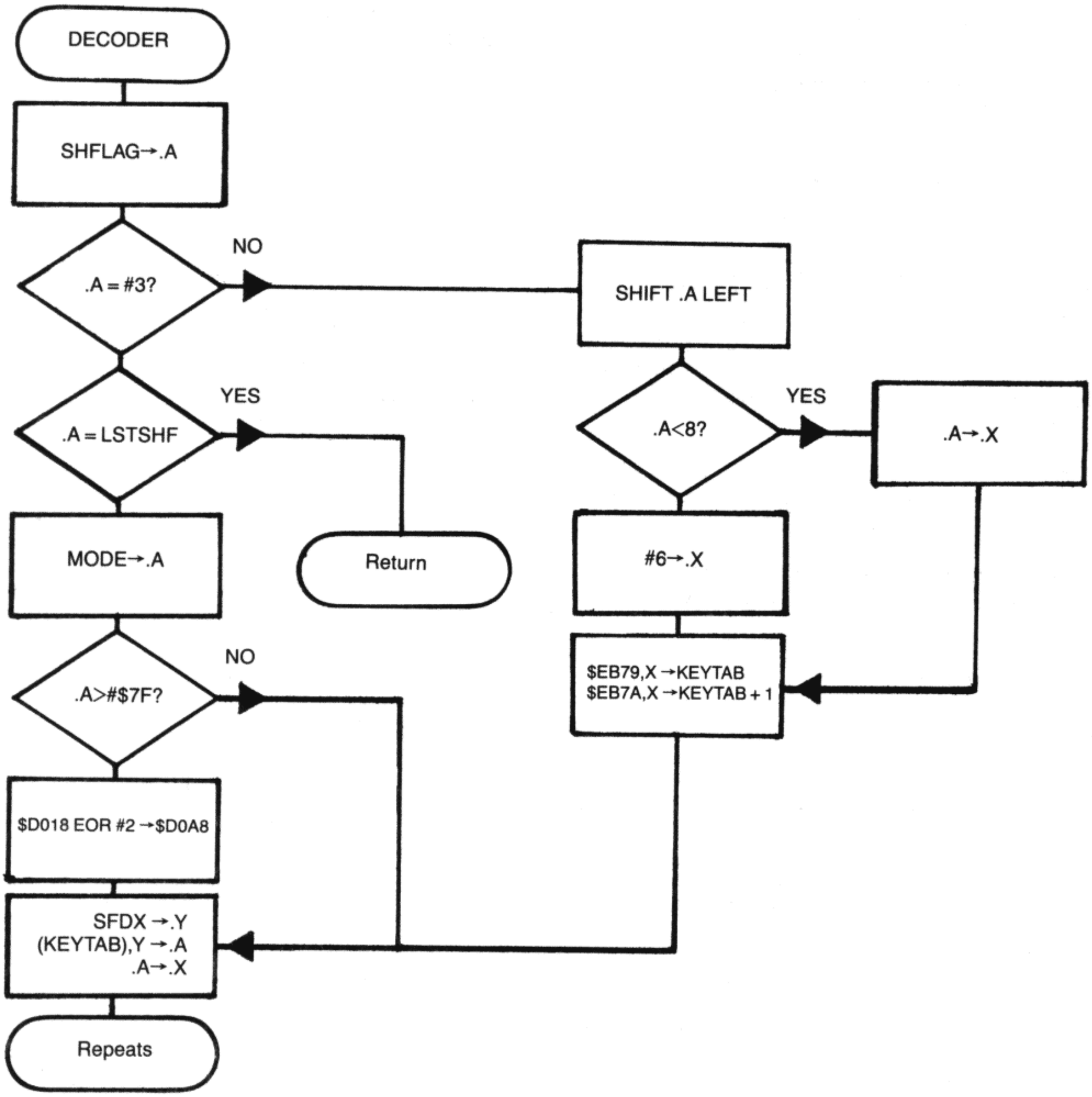
Now comes the part that some readers may not be aware of. After the scan is completed, a jump is made to the vector contained in KEYLOG. Normally KEYLOG contains $EB48 which is the start of the keyboard DECODER routine.

'SFDX' AND 'SHFLAG' ARE PASSED TO THE ROUTINE WHOSE ADDRESS IS CONTAINED IN 'KEYLOG'. THIS MAKES IT EASY TO RECONFIGURE THE KEYBOARD SHOULD YOU WISH TO DO SO, OR EVEN TO INTERCEPT AND ACT ON ANY PARTICULAR KEY, PROVIDED YOU OBEY THE RULES.

Change the vector to point to your own routine. When your routine has completed, jump back to the Kernel at the appropriate point which will depend on exactly what your routine is doing. Your routine must stay invisible to the Kernel, i.e. you must preserve the interface. The detailed flowcharts, the DECODER routine described below and a disassembled listing of the Kernel Keyboard routines should provide you with all the information you need. As an example, I have included a Program in this article to show how you may get a click out of each key pressed.



The DECODER Routine



This routine is entered at $EB48. SHFLAG and SFDX are used to compute a value in 'X' which in turn is used to fetch a table pointer from the table of addresses starting at location $EB79. The pointer is stored in KEYTAB and will point to one of four tables depending on the value of SHFLAG. The index in SFDX is then used to fetch the actual CHR$ value of the pressed key from the table pointed at by KEYTAB.

The four tables used are:

1. NORMAL	location $EB81
2. SHIFT	location $EBC2
3. COMMODORE	location $EC03
4. CONTROL	location $EC78

Each table is 65 bytes in length. The last byte is always = $FF, just in case no keys were found to be pressed (apart from those in SHFLAG). In this case SFDX will still contain its initial value of $40. $FF is also stored in all locations where no action is expected from a key. For example the CHR$ value for '*' in the CONTROL table is $FF. This is why you get no action when you press Control–*.

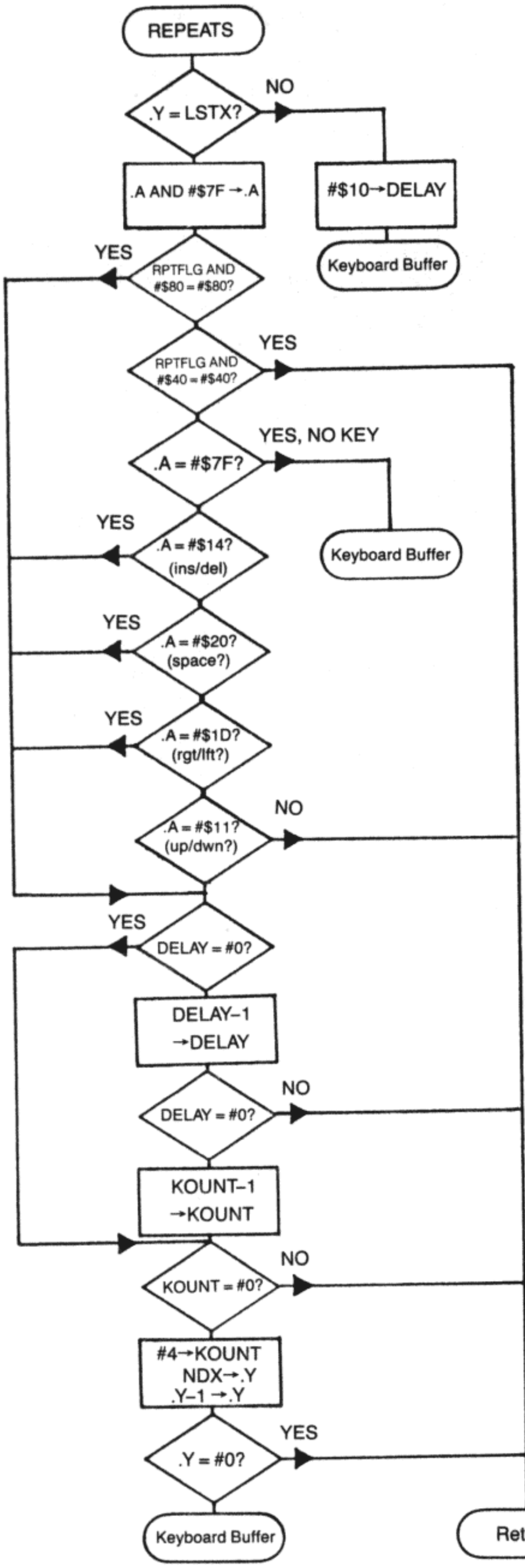
Also performed in this routine is the switch of character sets if SHFLAG is found to contain the value '$03' (i.e. Shift and Commodore pressed together). In this case, LSTSHF is examined to see if it also contains '$03'. As LSTSHF holds the value of SHFLAG from the previous activation of the interrupt routine, this will mean that the switch has already been made. No further action is taken and a jump is made directly to the RETURN routine. If LSTSHF is not the same as SHFLAG, then the variable, MODE, is examined. If the most significant bit is set here, then the character set is switched – lower to upper case or vice versa. This is done by XOR'ing location $D018 with the value $02.

The CHR$ value fetched from the appropriate decode table is stored in register 'X'. A jump is then made to the REPEATS routine.

$FF IN THE DECODE TABLE SIGNIFIES A NULL–KEY WHICH MUST NOT BE INSERTED INTO THE KEYBOARD BUFFER.

'MODE' CAN BE MODIFIED TO DISABLE OR ENABLE THE SWITCHING OF CHARACTER SETS.

The REPEATS Routine



This routine is entered at location $EAE0. SFDX is compared with LSTX. LSTX holds the value of SFDX from the previous activation of the interrupt routine. Therefore if SFDX is different to LSTX, it will mean that this is a new key–press. In this case, a value of $10 is written to DELAY. DELAY is a count of the number of activations of the interrupt driver before which a new key–press may be repeated. This accounts for the noticeable delay before which a key starts to repeat. If there was no delay it would be impossible to use the keyboard.

For a new key–press, a jump is made to the KEYBOARD BUFFER routine. Otherwise RPTFLG is examined. If the most significant bit (bit '7') is set, then all keys will be repeated. If not, then if bit '6' is set, all repeats are disabled and a jump is made to the RETURN routine. For all other values of RPTFLG, only the normally repeating keys are repeated, i.e. SPACE, INST/DEL etc.

A key is repeated only when both DELAY and KOUNT have been decremented to zero. DELAY is always set to $10 when a key is initially pressed. It is decremented once per interrupt activation until it has reached zero. Then the same process is repeated for KOUNT. When KOUNT has reached zero, then it is time to repeat the key. KOUNT is reinitialized to a value of 4 and a jump is made to the KEYBOARD BUFFER routine if there are no keys currently in the keyboard buffer, i.e. if NDX = '0'. For all other cases, the REPEATS routine jumps directly to the RETURN routine.

RPTFLG MAY BE MODIFIED TO TURN ON OR OFF THE REPEAT PROCESS.

Get A Click Out Of Your Keyboard :

This is an example of a way to plug your own routine into the interrupt code. The routine runs in the cassette buffer and is initialized by doing a SYS to location 828 ($033C).

Basic Loader:

```
10 rem get a click out of the keyboard
15 rem this is done by changing the table setup vector
20 rem – keylog – at location 655 ($28f) to enter the click
30 rem routine situated in the cassette buffer. if a new
40 rem key press is detected, we prod sid to click his heels.
60 for i = 828 to 865 : read a : poke i,a : next
70 sys 828
100 data 120, 169,  73, 141, 143,   2, 169,   3, 141, 144
110 data   2,  88,  96, 165, 203, 197, 197, 240,  15, 162
115 data  40, 169,  15, 141,  24, 212, 202, 208, 253, 169
125 data   0, 141,  24, 212,  76,  72, 235,   0,  –1
```

Assembler Code:

```
*       =   $33C      ;Cassette Buffer
        sei           ;Disable Interrupts
        lda   #<click ;Change KEYLOG
        sta   $28f    ; to point to
        lda   #>click ; the
        sta   $290    ; Click routine
        cli           ;Enable interrupts
        rts           ;Return to Basic
click   lda   $cb     ;Don't click
        cmp   $c5     ; if SFDX
        beq   clend   ; equals LSTX
        ldx   #40     ;Click Counter
        lda   #15     ;Thanks JEFF GOEBEL
        sta   54296   ; for the Click Poke
cloop   dex           ;Timeout
        bne   cloop   ; on click count
        lda   #0      ;Turn Click
        sta   54296   ; off
clend   jmp   $eb48   ;Return to interrupt
```

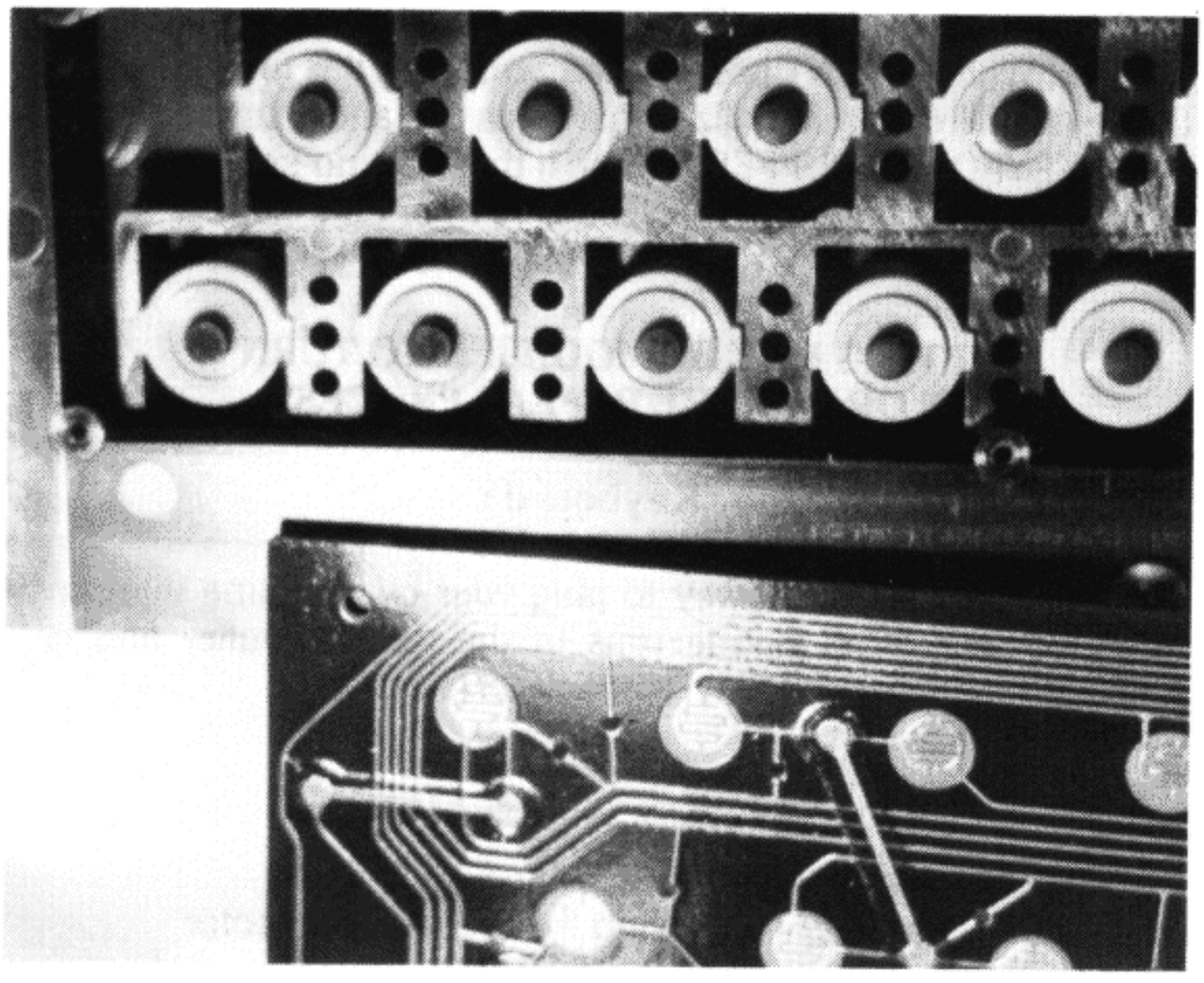
Conclusion

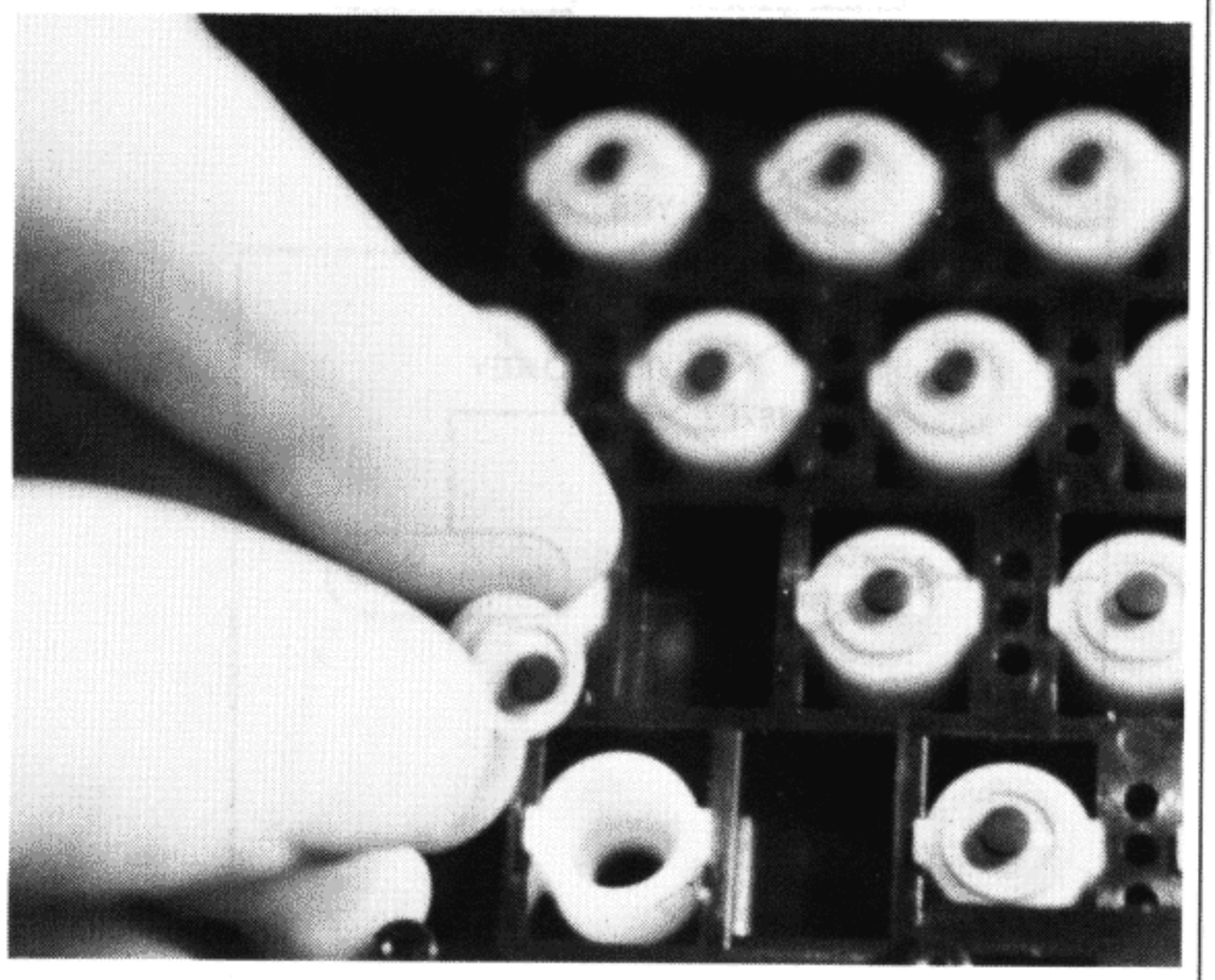
This completes the description of the Kernel Keyboard Driver. I hope it has proved instructive and provided enough information for those of you who wish to manipulate the driver to suit your own particular needs. In Part 2, we will develop our own driver, based on key–changes rather than key–presses. This will truly be versatile enough to suite anybody's needs, while still preserving the standard interface to BASIC and other applications that currently use the Kernel Driver for their keyboard handling.

Fixing Commodore Keyboards

Harold Anderson
Oakville, Ont.

Repair your failing keyboard yourself





Most people who have an aging PET have by now encountered problems with the keyboard. The problems usually start with double or triple letters being generated every time a key is hit. Eventually the situation deteriorates to the point where the key does not generate a response at all. The keys that go first are usually those most commonly used, such as the shift, space, and return keys. If you have kids playing games on your computer, the keys used in the game will soon give trouble. The letter 'A' seems to go very quickly, since it is the "gun" in a lot of games.

Let me explain what is happening here. The keyboard of virtually any computer is really a collection of switches, one for each key. On the PET and Commodore 64, the computer looks at these switches 60 times a second to see if they are closed. As soon as the computer sees that a switch is closed, it prints a letter. The letter will not be printed again unless the switch is opened and reclosed. (Exceptions are the space and cursor–control keys, which have an auto–repeat feature.)

If the connection made by the keyswitch is intermittent, the computer will think that it has been opened and closed several times even though you have only pushed it once.

There are several different types of switches used in computer keyboards. The type that Commodore uses are made with electrically conducting rubber. Figure one shoes part of a disassembled PET keyboard. The bottom of the picture shows the contact board which has been removed from the underside of the keyboard assembly. You will notice that it is covered with patterns that look like interlocking fingers. These fingers are made of gold–plated copper and form the two poles of the switch. The top of the picture shows a collection of white circles with black dots at the center. The white circles are the bottom ends of the key plungers and the black dots are conducting rubber pads. When the keyboard is assembled, the contact board is turned over and screwed to the underside of the keyboard assembly so that the rubber pads sit about 1/8 of an inch away from the gold–plated fingers. When a key is pressed, the conducting rubber pad moves down to touch the gold fingers and short–circuits them together, thus closing the switch.

The keyboard on a Commodore 64 works on the same principle except that two gold–plated "contacts" about 1/4 of an inch apart are used instead of interlocking fingers. The piece of rubber which short circuits the gold contacts is shaped a bit like a dog bone and the ends of the bone press on the contacts when the key is pressed. This modified design has been used because it allows conductors to other key switches on the contact board to run between the two contacts. This simplifies the fabrication of the contact board.

When the keyboard starts to give trouble, two things are happening. The first is that the rubber pad and the gold–plated surfaces are getting covered with dust which insulates the surfaces. The other problem is that after long use, the surface of the rubber itself seems to lose its ability to conduct electricity. Both of these problems are easy to fix once the keyboard is opened up.

Before you can service the keyboard, you have to get at it. Start by unplugging the computer. On PETs the whole top of the computer can be swung up. The screws that hold the top down are at each end of the keyboard under the lip of the white upper cover for the computer. On the Commodore 64 there are three screws along the underside of the front edge of the computer. In both cases the front of the computer can be lifted up when these screws are removed. On the Commodore 64 it will be necessary to unplug the wires to the red pilot light as soon as the cover is lifted. The plug is just below the flap on the metalized cardboard shield that covers the main circuit board. Notice which way around this plug goes so that

you can put it back properly.

When you look at the bottom of the keyboard you will see that the contact board is held to the bottom by about 20 tiny screws. Before taking the screws out, you will have to remove the two wires that go to the shift lock switch (right below the shift lock key). You will need a soldering iron to do this. If a soldering iron is a hard item to come by, I would suggest that you extend the wires on the switch so that in future the contact board can be removed without unsoldering the wires.

With the bottom of the keyboard off, you can clean off the offending components. The pads and gold surfaces can usually be cleaned by brushing them with a clean, dry paintbrush. If you smoke a lot, you may have a film of tar over the surfaces. I would suggest that you remove it with methal alcohol on a clean rag.

If the keyboard still gives trouble, the surface of the rubber pad can be renewed by using VERY fine sandpaper on them. (400 grit sandpaper of the type used for wet sanding in body shops seems to work well.) Sand them just enough to make the surface of the rubber dull. In order to sand them you will find it easier to pop the pads out, as shown in figure 2, and stick them over the square, unsharpened end of a pencil. It would be fairly tedious to do this to all of the keys so I suggest that you do it only to the keys giving trouble. If you can't get the sandpaper, swap the pads on the ends of the key plungers so that the troublesome pads are on seldom–used keys like Z and X. The pads can also be purchased as a spare part for about a dollar each.

Although this sounds complicated, a keyboard can be repaired in about 1/2 an hour, which definitely beats living with one that gives trouble.

Life With the 1541

Michael Quigley
Vancouver, B.C.

Do you hate your 1541 disk drive with a passion? I mean – do you leave it sitting in the front seat of your car with the windows open, the doors unlocked and a large sign attached reading "STEAL ME"? If so, welcome to the club. I'M sure if Benjamin Franklin were alive today, he'd revise one of his most famous quotes to read: "In this world, nothing is certain but death and taxes. . . and the 1541 developing problems."

Probably the biggest problem of the 1541 is "going out of alignment". What this means is that the read/write head is unable to correctly find information on the disk. A major indication of trouble is the red LED on the front of the drive flickering when attempting to read disks, particularly those not formatted on your own drive, or disks which you made several months ago.

The cause of all this is relatively simple. The drive contains a stepper motor which advances the head from one track to the next; no easy task, since the tracks are a fraction of an inch apart. The stepper motor shaft goes up through a pulley which is connected to a metal band, which in turn is attached to the read/write head. Whenever a new disk is formatted or – worse yet, an error is detected on a "copy protected" disk – the pulley knocks up against a head stop which is at the hypothetical track zero. This is the source of the rattling noise which occurs when a disk is "NEWed".

After X number of knocks against the head stop, the shaft in the middle of the pulley begins to slip ever so slightly, since the two are not permanently attached to each other. The result is that the read/write head slips out of position as well. There is a theory that the heat produced by the drive (which is considerable) may also contribute to this misalignment.

The cures for this malady – aside from using the drive as a speed bump in the alley behind your house – are varied. An alignment disk and oscilloscope are necessary for precision. In some cases, moving the stepper motor slightly after loosening the screws which hold it to the bottom of the drive may be sufficient. In more extreme cases, a notch has to be carved in the top of the stepper motor shaft to serve as a screw–like slot where the pulley is manually moved back to its correct location.

Other solutions of a permanent nature (after aligning the disk, of course) include using Krazy Glue or some such epoxy substance to hold the pulley to the shaft. This may be done in combination with drilling a hole through the pulley to the central shaft and inserting a pin. (Such a precision job must be done with care, because pieces of metal can find their way down to the inside of the stepper motor, rendering it totally useless.) It should be pointed out that any of these actions will void your warranty, if it's still in effect.

Editor's note: While the 1541 does have the bug of occasionally going a bit out of alignment, on the whole it has features that can't be found on much more expensive drives. It's not the fastest thing in the world, but it's extensive ROM code makes it fairly "smart", and gives it lots of features and flexibility. It's ironic, perhaps, to state this after the above article, but the 1541 drive may be one of the best deals for the money in the microcomputing market. –T. Ed.

Learning The Language Of DOS

Richard T. Evers, Editor

In this world there are many great mysteries, one of which is the Commodore Disk Operating System (DOS). A disk drive is for most a magical container in which diskettes are inserted to store vast quantities of data, and retrieve the same with unerring accuracy. There are but a few beings worldwide who fully understand the teachings of DOS, but alas, few have the power of written communications at their access. Today, we will embark on a journey into the deep recesses of DOS, to gradually bring about an understanding of how it works, and why it is superior to others.

Commodore DOS does not require 'booting up' before access to the drive can begin. Unlike other manufacturers' computer systems, Commodore DOS is held in ROM, and is always with us. Once powered up, DOS comes alive, and with it numerous avenues of disk drive power are exposed. Within the drive, knowledge is found. Commodore had the mental fortitude to incorporate 'intelligence' into their drives, therefore they come complete with microprocessor(s), various IC's, a fair quantity of RAM, and a large chunk of ROM.

Before delving into the inner commands required to help you communicate with DOS, let's get into a bit of theory.

A Bit Of Theory

All DOS communications occur through the Command channel; channel 15. To recap, the incantation required to incite channel 15 is:

```
OPEN LF,UA,15
```

where LF represents the logical file address (1–127), UA represents the unit address of the drive, usually 8, and finally, 15 is the secondary address, which is mandatory.

Once this communications line has been set up, full duplex communications can begin. In the sub–sections soon to follow, all the special disk commands will be covered correctly, so you can learn to speak DOS, therefore bringing you closer to your DOS. Till then, more theory is required.

More Theory

Once the 15th channel has been OPENed, communications can begin, as I have stated in the preceding paragraph. Among DOS sleuths, the Command channel is often referred as the Command Buffer, CMDBUF. For the 1541 and 2031LP, this is held at $0200–$0229 in RAM. The 4040, 8050, and 8250 have it tucked away at $4300–$433A. The command buffer to the DOS is like the input buffer to your computer. Everything of importance goes through this area, is deciphered, then acted upon. All commands, filenames, and special characters, or simply, whatever it takes to make the DOS stand up and listen.

Whenever an error has occurred with the drive, it again is controlled through the 15th channel. This error can be read from the drive via INPUT# or GET# statements, but it can also be PEEKed, via a disk equivalent, from RAM. The error buffer in the 1541/2031LP is at $02D6–$02F8, and can be found at locations $43DC–$43FF for the 4040/8050/8250 dual units.

Once the DOS receives your command input, it breaks it down into simple terms that can be readily acted upon. If you wanted to do a READ from disk, all the necessary RAM locations would be set up first by the Interface Processor (IP), then the value of $80 would be put in the JOB QUE. From this point the system would wait for the Floppy Disk Controller (FDC) to come along and discover the job waiting on the JOB QUE. Once discovered, the FDC would kick in and do a READ. In this way, whatever you tell your drive to do can be handled quickly and smoothly with co–operation within.

The JOB QUE, as described above, is the special spot within the drive that informs the FDC what to do. In past, Commodore put a couple of microprocessors in their drive units, namely the 4040, 8050, and 8250, for the purpose of handling different functions. The first processor, the Interface Processor (IP), is responsible for accepting commands via the 15th channel, and deciphering what they actually mean. From here, the IP sets up all the necessary conditions for the Floppy Disk Controller (FDC), and stores the appropriate job code on the JOB QUE. Every 10 ms. the JOB QUE is scanned by the FDC, just to see if any action is required. If not, the FDC goes back through its rounds doing any housekeeping chores required. In Commodore lingo, this is referred to as going into the IDLE loop, waiting for something to do.

If a job is encountered on the JOB QUE, the FDC takes it and acts upon it. Once the job has been completed, correctly or incorrectly, the FDC finds the appropriate error code to describe the action performed, and stores that over top of the original job code in the JOB QUE. In this way the FDC can let the IP know that the job has been completed, or has run into a few snags along the way. From here, the IP takes the error code and stores the corresponding description in the error buffer. A clean and neat solution.

As stated initially, Commodore at one time put a few microprocessors in their drive units. The 4040 has one 6502 and one 6504. Both the 8050 and 8250 have twin 6502's. But then came the days of the 1541 and 2031 LP, and the microprocessor count went down to one, a single 6502. Commodore found that with the correct coding, one severely overworked processor could do the job of two. A question remains for most: Why overwork and therefore slow down the system with one processor, when two has been proven in past? There is Commodore logic working here.

In order to bring the price of the units down to a reasonable level, Commodore got smart. The larger drives were, and still are, very expensive, if you can find them. The newer drives were made to be affordable. Specifically, the 1541 was made to be affordable. No longer does it host the expensive, but reliable, IEEE port. It's been replaced by the Serial Port. Though this does slow down access to the drive considerably, (63 seconds to LOAD a 100 block program from a 1541 into a C64 vs. 16 seconds for the same from an 8250 into an 8032), it does help keep the price low.

Another method used to drop the price tag was to decrease the quality of the stepper motor and stop mechanism. For this reason, you will find that the 1541s and 2031s require service more often due to the stop mechanism knocking the alignment out. Ever hear that familiar GRIND GLICK coming from your 1541, and wondered what was happening? It's the drive trying to compensate for a bad diskette or a stepper motor that is slightly out of whack by throwing a BUMP command on the JOB QUE and telling the FDC to BUMP the head against the back stop. This is how DOS deals with the possibility that the head carriage has jumped out of the groove in the head positioning mechanism. The stepper motor swings into action thus slam-dancing itself up against the stop mechanism numerous times, creating the horrible noise. When ever this noise occurs, there is a chance that your drive is not so gently being thrown out of whack. Remember that the next time you consider a disk protected package. Due to the inherent READ errors to be encountered on the diskette surface, the BUMP command will be used alot. That disk protected package you just bought could cost you in the long run at the repair shop.

For those of you who are interested, the JOB QUE is located at $1003–$100F for the 4040/8050/8250 drive units, and $0000–$0005 for the 1541/2031LP. Below is a chart describing the appropriate job number and its relative description. Please exercise caution before using the JOB QUE though. RAM pointers necessary for the correct execution of your wishes should be set up CORRECTLY before placing any job number on the JOB QUE. When using the JOB QUE, you are literally bypassing the drive's built in protective mechanisms. Therefore, one wrong move and you may find yourself trying to pry the head away from the inside of the drive casing.

FDC Job Codes

Code	Operation	Description
$80	FDC READ	– read data from diskette
$90	FDC WRITE	– write data to diskette
$A0	FDC VERIFY	– verify written data
$B0	FDC SEEK	– look for a specific sector
$C0	FDC BUMP	– bump the head for correction
$D0	FDC JUMP	– jump to a machine language routine
$E0	FDC EXECUTE	– execute a machine language routine

FDC Error Number Returned On The Job Que

Code	Error	Description
$01	OK	
$02	READ ERROR	– can't find block header
$03	READ ERROR	– no sync character
$04	READ ERROR	– data block not present
$05	READ ERROR	– checksum error in data block
$06	READ ERROR	– byte decoding error
$07	WRITE ERROR	– write/verify error
$08	WRITE PROTECT ON	– write with write protect on
$09	READ ERROR	– checksum error in header block
$0A	READ ERROR	– data extends into next block
$0B	READ ERROR	– disk ID mismatch

With that chart complete, it's time to move on, deeper into the commands that will allow you to communicate with your DOS. The information just covered is rather high level. Therefore, the following commands should be mastered before considering working directly with the FDC.

Conventions

A few short forms will be used when describing access to the DOS. These short forms are really descriptions for numeric values necessary for the correct execution of the command by DOS :

AH	= memory address high byte	0–255
AL	= memory address low byte	0–255
CH	= channel – used in lieu of LF to prevent confusion later	
DR	= drive number	0 or 1
LF	= logical file number	1–127
NC	= number of characters	1–255
P	= desired position within buffer	0–255
SA	= file secondary address number	2–14, 15=Cmd Chan
S	= sector	0–max
T	= track	1–max
UA	= drive unit address	Usually 8

It will also be assumed, as stated earlier, that the command channel is OPEN before attempting to send commands over the bus.

B–A : Block–Allocate

On your diskette lies an important block of data. This is the Block Availability Map, or simply stated, the BAM. The BAM is responsible for managing the whereabouts of all your files, and making sure that you do not write over the same block of information twice. The BAM is a bit map of each and every track and sector on your diskette and, with the proper amount of knowledge, allows for a complete breakdown of how the diskette has been allocated. Time for an explanation.

On a portion of track 18, sector 0 for the 1541, 2031, and 4040 drives, lies the BAM. Due to the limited capacity of these drives, the BAM is small. The 8050 has a BAM that consumes 2 complete blocks, track 38 sectors 0 and 3, where the 8250 has a BAM of 4 block length, track 38 sectors 0, 3, 6, and 9. For each, the concept of BAM is similar.

The 1541/2031/4040 drives use 4 bytes per track in the BAM. The larger drives have five bytes. The first byte of each carries the

current count of blocks free on that track. The next 3 or 4 bytes carry a bit map of the track. Take for example the following 4 bytes of data:

15 FF FF 1F

This has been taken from the BAM of a freshly NEWed 1541 diskette. Please remember that these figures are in hexadecimal. The 15 represents the number of blocks free on that track, which is on track 1 in this case. Hexadecimal 15 = 21 decimal, therefore there are 21 blocks free on this track. The next 3 bytes (4 bytes if taken from an 8050 or 8250 diskette) represent a bit map of the allocation state of the sectors on this track. To figure this out, you have to think of everything in binary.

FF = 11111111 : FF = 11111111 : 1F = 00011111

Now, since Commodore has always adapted a low byte/high byte strategy on everything they do, the BAM is laid out the same way. Therefore,

FF FF 1F

should really be read as

1F FF FF

or in binary,

000111111111111111111111

A bit ON (1) represents a free sector, a bit OFF (0) represents one that has been allocated. In this case, there are a maximum of 21 sectors available on track 1 of a 1541/2031/4040 diskette therefore, the last 3 bits have to show as allocated. They don't exist and can never be written to.

The reason the larger drives have a four byte bit map instead of three is because they are double density, double tracking drives, and allow for a greater number of tracks per diskette, and a greater number of sectors per track. The maximum number of sectors per track for these drives is 29, therefore it cannot be represented with only three bytes (24 bits). Four bytes equal 32 bits, which is just perfect for the larger drives. To better understand the physical sector per track distribution on the various diskette surfaces, check the chart below :

	No. of Sectors				No. of Sectors	
Track #	1541	2031	4040	Track #	8050	8250
01–17	21	21	21	1– 39	29	29
18–24	19	19	19	40– 53	27	27
25–30	18	18	18	54– 64	25	25
31–35	17	17	17	65– 77	23	23
				78–116	na	29
				117–130	na	27
				131–141	na	25
				142–154	na	23

With that explained, we can get back on track and explain the Block–Allocate command.

As the name implies, you can tell the DOS to allocate a specific block whenever you choose. The format is:

print#LF, " b–a: " DR;T;S

. . .then: close LF

This will automatically update the BAM, but will also CLOSE every other file currently OPEN due to the closing of the command channel.

With some drives, ie. 1541, 2031LP, and a few of the older ones, Block–Allocate does not work correctly. It really is best if you read the BAM into RAM (disk or computer), update it manually, then Block–Write it back again. This will save many problems, and also allow you to do so without closing down the 15th channel. More on Block–Write later.

B–F : Block–Free

For every action there is an equal and opposite re–action just waiting for its chance. Block–Free is like Block–Allocate in reverse. It has the same format, but will de–allocate any block you choose, provided the block was allocated before. As before, this command has been found to be terminally ill in a few of the drives, as mentioned above, and cannot be trusted. Block–Read the BAM into disk RAM, modify it there or move it into the computers memory for modification, then Block–Write it back in again. Before doing so, though, please read all about Block–Read and Block–Write. They too are terminally ill, therefore their equivalent counterparts are best used. More on that now.

B–R or U1 : Block–Read

You already know that Block–Read and Block–Write are very unreliable. Instead, the U1 and U2 commands should always be substituted.

U1, along with the rest of the User family, are terrific to work with, and will not give you problems. U1 will read a specific block of your choosing from the disk surface into disk RAM. If set up correctly first, you can even make sure that it is read into the RAM of your choosing.

Once the data has been moved into RAM, you can set the Buffer–Pointer to point at the spot in the buffer you want to start reading from, then begin the retrieve process. The default on the Buffer–Pointer is to start you at the beginning of the block.

A technique many employ when reading, updating, and writing data directly to disk, is to update disk RAM, instead of bringing it into computer RAM for update. It is faster, and less prone to error. It is also easier to update a single character of data this way than any other.

To help you out in this department, I have prepared a chart of all the RAM buffers within the drive units, so you know which buffer OPENed corresponds to which address in RAM (top of next page).

From the chart, you can now deduce that the command:

open CH,UA,SA, " #0 "

. . .would allow you to read data directly into either $0300–$03FF with the single drives, or $1100–$11FF with the dual units. Assigning an actual number after the '#' in the statement allows you to pick and choose exactly which buffer you want. A nice option. Please note the use of CH (channel) instead of LF (logical file) in this example. They are the same, but will help avoid confusion later.

1541/2031LP RAM Buffer Layout

$0300–$03FF : Buffer #0
$0400–$04FF : Buffer #1
$0500–$05FF : Buffer #2
$0600–$06FF : Buffer #3

4040/8050/8250 RAM Buffer Layout

$1100–$11FF : Buffer #0
$1200–$12FF : Buffer #1
$1300–$13FF : Buffer #2
$2000–$20FF : Buffer #3
$2100–$21FF : Buffer #4
$2200–$22FF : Buffer #5
$2300–$23FF : Buffer #6
$3000–$30FF : Buffer #7
$3100–$31FF : Buffer #8
$3200–$32FF : Buffer #9
$3300–$33FF : Buffer #10
$4000–$40FF : Buffer #11

The format of Block Read, User style, is:

```
print#15, " u1: " ;CH;DR;T;S
```

When this command is executed, the track and sector from the drive number specified will be read into buffer #0. From here, you are free to do as you please.

B–W or U2 : Block–Write

B–W is even more untrustworthy than B–R in the 1541, 2031LP and 4040 DOS, and the U2 command is the easiest way to circumvent problems. Do everyone a favour and forget that Block–Write (b–w) exists, and stick with the proven contender. Your programs will be happier.

As with Block–Read, Block–Write is the flip side of the coin. Set up in exactly the same format as it's brother U1, U2 will write to diskette the contents of the buffer that you specify. You can print to that buffer, then write it with U2, or you can manually whiz about within the RAM with the Memory commands leading the way. Your choice. The format is:

```
print#15, " u2: " ;CH;DR;T;S
```

Buffer–Pointer :

This one's not diseased. Buffer–Pointer allows you to point exactly where you want to start reading from, or writing to, within a specific buffer of RAM. The default of this one is set by DOS, and is at the start of the buffer. You can choose wherever you want though, from location 1, the start, to location 255, the end. A terrific animal to have if you know what will be located in the buffer prior to reading it in. The format is:

```
print#15, " b–p: " CH;P
```

B–E : Block–Execute

A command that has found limited use to date, but should be covered for this issue, at least on a cursory level.

Block–Execute will allow you to pick a specific track and sector from disk, down load it from diskette into disk RAM, then execute it where it sits. The reason for its limited appeal is because until lately, little has ever been publicly released about the DOS so not too many inner-disk programs have been written. Commodore, for whatever obtuse reason they have, has never really told anyone about the inner workings of their drives. Sure, everyone can find info on the diskette layouts, and can also find a list of commands, but as far as ROM routines go, forget it. For anyone who is interested, we will be releasing our Reference Book soon, and held within its many pages will be RAM/ROM maps for most of the drive units. The 1541 Zero Page RAM Map at the end of this article is just a sample of what you'll find.

The format for Block–Execute is:

```
print#LF, " b–e: " ;CH;DR;T;S
```

The Memory Commands

The true spirit of disk programming comes into play when dealing with the memory commands. With these commands, you have at your access total control of your drive. At your slightest whim, you can decide the fate of your drive, be it to destroy itself trying to follow your wishes senselessly, or to execute a well researched string of commands that will bring about fabulous results. Careful use of the Memory commands can unleash the true power of the DOS.

M–R : Memory–Read

Similar to the PEEK command in BASIC, Memory–Read will allow you to read data from anywhere within the drive. You specify the address to read from, and the drive gets the info for you. Terrific. The format is:

```
print#15, " m–r " chr$(AL)chr$(AH)chr$(NC);
```

then. . . get#15,a$

The NC, number of characters, is an optional parameter, and does not have to be used. If it is, you can specify exactly how many characters you would like to read instead of a separate M–R for each single byte.

Memory–Read can be used as a substitute for Block–Read if you know the address of the buffer that contains the desired data. You can specify exactly where you would like to read data from and start reading. It's like a manually executed Buffer–Pointer. Then a GET# through channel 15 will retrieve the data.

One final use for Memory–Read is to discover what makes your disk unit tick. To write a routine to read through disk memory and return the results to the screen takes little effort, as demonstrated below :

```
10 hx$ = " 0123456789abcdef " :z$ = chr$(0):flag = 0
15 open 15,8,15
20 input " start, end " ;s,e
25 for lp = s to e step 8
30 ah% = lp/256 : al = lp–ah%*256
35 flag = 1 : va = ah% : gosub55 : print ht$; : va = al
     : gosub55 : printht$ "  " ; : flag = 0
```

```
40 print#15, " m-r " chr$(al)chr$(ah%)chr$(8)
45 for in = 0 to 7
50 get#15,a$ : va = asc(a$ + z$) : v$ = v$ + chr$(va or 64)
55 h% = va/16 : l = va-h%*16
60 ht$ = mid$(hx$,h% + 1,1) + mid$(hx$,l + 1,1)
     : if flag then return
65 print ht$ "   ";
70 next in
75 print v$ : v$ = " "
80 next lp
85 goto 20
```

A clean and neat method to scoot about within your drive and discover all that lies in wait for you. Of course, the routine above can be made more useful. For a larger version of the same, check the article 'Drive Peeker' in this issue. It's a little more versatile for the user.

M-W : Memory-Write

The hero of the memory commands, and one in which the serious disk programmer will use far more than anything else. This single command allows you to place your own thoughts anywhere you please in the drive, RAM permitting, for the purpose of future execution, or for the simple purpose of changing the drives characteristics. In order to manually bypass the normal operating system of the drive, and tell the FDC to spring to life, Memory-Write is the command of choice.

As stated above, many characteristics of your drive can be altered by a few well placed writes - the unit address can be altered, the JOB QUE can be loaded, and with the proper amount of research, most of the drive formatting characteristics can be modified for an originally designed diskette. The designer diskette, a novel idea for the illustrious programmer. If this tickles your fancy, wait for our Reference Book. It will be worth the wait.

The format of Memory-Write is:

```
print#15, " m-w " chr$(AL)chr$(AH)chr$(NC);chr$(data)
```

M-E : Memory-Execute

As a final compliment to Memory-Write, Memory-Execute will execute whatever code you want within the drive. Point it your own code and watch it turn your thoughts into realities. Point at Commodores code, and see if they will work for you. Whatever you care to do, it's available and willing to go.

Some people argue that there is no 'safe' room inside of a drive. This is true, if you're not in control of your drive. When placing code in disk RAM, pick a buffer that would be the last one used, ie. the highest buffer number. For everyone out there with the single drives, this may be a tall order. There are only a few buffers, and it will be hard to make sure your code doesn't get stepped on. Remember to allow as few files open at the same time as possible, and your code may be safe. Otherwise, there are a few spots held deep within RAM that could be used as a temporary hiding place, providing your code is small.

These locations are all available for use, as long as you're not using RELative files with your particular application. There are quite a few single byte areas, and again, quite a few blocks of RAM tucked away, but unless the exact situation is known that it will be used with, it might be best to leave them alone.

Single	Dual	Label	Description
00B5-00BA	0059-0060	RECL	Low Rec# To Find Rel File
00BB-00C0	0061-0068	RECH	High Rec# To Find Rel File
00C1-00C6	0069-0070	NR	Next Relative Record Table
00C7-00CC	0071-0078	RS	Relative Record Size Table
00CD-00D2	0079-0080	SS	Side Sector Table
0104-01FF	0100-01FF		The Stack (use cautiously)

To get back on track, the format of Memory-Execute is:

```
print#15, " m-e " chr$(AL)chr$(AH)
```

The User Commands

The User series of commands fall into three categories. The first, U1 and U2, come under the heading of diskette access. The second category, drive housekeeping, encompasses the U0, U9, and U: commands. They take care of internal drive stuff that keep the drive content. The third and final category is drive access commands, U2-U8, which go hand in hand with the Memory commands discussed earlier. Take a look below for a table of all the User commands.

Standard Syntax	Alternate Syntax	Function
U0		Reset of USR Jump Vector In Disk RAM
U1	UA	Block-Read Replacement
U2	UB	Block-Write Replacement
U3	UC	Jump To $1300(dual) or $0500(single)
U4	UD	Jump To $1303(dual) or $0503(single)
U5	UE	Jump To $1306(dual) or $0506(single)
U6	UF	Jump To $1309(dual) or $0509(single)
U7	UG	Jump To $130C(dual) or $050C(single)
U8	UH	Jump To $130F(dual) or $050F(single)
U9	UI	Jump To NMI : $10F0(dual) or $FF01(single)
U:	UJ	Power Up Vector (system reset)

There are bugs in most of these commands in the 1541/2031LP drives. The alternate syntax of each User command can give unexpected results - avoid them like the plague. Use U0-U: for your work, and you should run into few difficulties. And now, a quick explanation for each command.

U0 is a command that has been poorly documented by Commodore in past. They have prepared charts on the User commands, always forgetting to included this one on it. One single sentence was donated once in an old disk manual. If

```
print#LF, " u0 "
```

. . .is executed, the USER JUMP vector in disk zero page will be reset to normal. Not a terribly useful feature, but one that will come in handy if you change the vector for something and need a quick way to return it back to normal.

U1 and U2, as mentioned previously, are replacements for Block-Read and Block-Write.

U3 through U8 are commands implemented to allow for a structured method of disk access adaptable for all machines. To use

these commands, you have to set up the RAM jump vectors first. Each vector is three bytes apart, therefore, you are expected to write in a JMP ($4C), then the lo/hi address of the code you intend to execute. Thereafter, you will be able to access your routine by a single execution of the User command. A fairly handy system that has seen little use in the past. The format for execution is:

```
print#LF, " uX "
```

. . .where X is the number of your choosing.

U9 is odd, but possibly useful. It jumps to the NMI vector, which in turn is like a system reset, without the flashing LEDs (power on diagnostics). The format is the same as before, X=9.

U: (*U - colon*) is a "power-on" system reset – handy for reseting the drive without physically powering down. If you want to make special internal code disappear, U: is the answer. If you have messed up badly inside of the unit, U: again. Sometimes, as I have found a few times in the past, you can mess up RAM so bad inside of your drive that even a system reset won't work. Then the switch at the back is the only alternative. Hopefully, U: will be sufficient for your needs.

That's All

And so ends this rather long but possibly informative article on the inner world of DOS. With a little belligerence and practice you can perform tricks inside DOS that will never be implemented as a BASIC command. It is for this reason that I suggest you don't include these tricks in "transportable" software. Future Commodore Disk Operating Systems may not recognize old tricks, however tried and true.

There is still quite a bit to be learned about DOS, for it is a very complex system. We at The Transactor are learning new facts almost daily, even though the 4040 has been with us now for well over 3 years. As we learn more, you will too. So, until we meet again. . .

1541 RAM Memory Map with Zero Page Contents at Power Up

references to Drive 1 are mostly unused locations

Hex Location		Content	CBM Label	Function
00-05	00	00	JOBS	Job Que: Buffer #0
	01	00		Buffer #1
	02	00		Buffer #2
	03	00		Buffer #3
	04	00		Buffer #4
	05	00		Buffer #5
06-11	06	00	HDRS	Job Headers: Buffer #0 - Low
	07	00		Buffer #0 - High
	08	00		Buffer #1 - Low
	09	00		Buffer #1 - High
	0A	00		Buffer #2 - Low
	0B	00		Buffer #2 - High
	0C	00		Buffer #3 - Low
	0D	00		Buffer #3 - High
	0E	00		Buffer #4 - Low
	0F	00		Buffer #4 - High
	10	00		Buffer #5 - Low
	11	00		Buffer #5 - High
12-15	12	00	DSKID	Master Copy Of Disk ID: Drive 0
	13	00		Drive 0
	14	00		Not Used - Drive 1
	15	00		Not Used - Drive 1
16-1A	16	00	HEADER	Image Of Last Header: ID Byte 1
	17	00		ID Byte 2
	18	00		Track
	19	00		Sector
	1A	00		Checksum
1B	1B	00	ACTJOB	Controllers Active Job
1C-1D	1C	01	WPSW	Write Protect Change Flag: Drive 0
	1D	01		Drive 1
1E-1F	1E	10	LWPT	Last State Of WP Switch: Drive 0
	1F	00		Drive 1
20	20	00	DRVST	Drives Current Status: Drive 0
21	21	00		Speed Timing Flag
22-23	22	00	DRVTRK	Drive Track Number: Drive 0
	23	00		Drive 1
24-2D	24	00	STAB	Storage Table For GCR Conversion
	25	00		
	26	00		
	27	00		
	28	00		
	29	00		
	2A	00		
	2B	00		
	2C	00		
	2D	00		
2E-2F	2E	00	SAVPNT	Temporary Save Pointer Location
	2F	00		
30-31	30	00	BUFPNT	Active Buffer Pointer
	31	00		
32-33	32	00	HDRPNT	Header Pointer: Track
	33	00		Sector
34	34	00	GCRPNT	GCR Pointer
35	35	00	GCRERR	Indicates GCR Decode Error
36	36	00	BYTCNT	Byte Counter For GCR/Binary Conv
37	37	00	BITCNT	Bit Counter
38	38	00	BID	Data Block ID
39	39	00	HBID	Header Block ID
3A	3A	00	CHKSUM	Checksum
3B	3B	00	HINIB	*not used directly
3C	3C	00	BYTE	*not used directly
3D	3D	00	DRIVE	Drive Number
3E	3E	FF	CDRIVE	Current Active Drive Number
3F	3F	00	JOBN	Current Job Number
40	40	00	TRACC	Track - Internal Storage Location
41	41	00	NXTJOB	Next Job
42	42	00	NXTRK	Next Track
43	43	00	SECTR	Sector Per Track For Formatting
44	44	00	WORK	Working Storage Location
45	45	00	JOB	Job Type
46	46	00	CTRACK	*not used directly
47	47	07	DBID	Data Block ID
48	48	00	ACLTIM	Accel Time Delay
49	49	39	SAVSP	Save Stack Pointer
4A	4A	00	STEPS	Steps To Desired Track
4B	4B	00	TMP	Temporary Storage Location
4C	4C	00	CSECT	Current Sector
4D	4D	00	NEXTS	Next Sector
4E	4E	00	NXTBF	Pointer To Next GCR Source Buffer
4F	4F	00	NXTPNT	Ptr To Next Byte Location In Buffer
50	50	00	GCRFLG	GCR/Binary Flag In Active Buffer
51	51	FF	FTNUM	Current Format Track
52-55	52	00	BTAB	Binary Table: GCR/Binary Work Area
	53	00		
	54	00		
	55	00		
56-5D	56	00	GTAB	GCR Table: GCR/Binary Work Area
	57	00		
	58	00		
	59	00		
	5A	00		
	5B	00		
	5C	00		
	5D	00		
5E	5E	04	AS	Number Of Steps To Accel With Head
5F	5F	04	AF	Acceleration Factor
60	60	00	ACLSTP	Steps To Go Before Complete
61	61	00	RSTEPS	Number Of Run Steps
62-63	62	05	NXTST	Pointer To Stepping Rtn - $FA05
	63	FA		
64	64	C8	MINSTP	Minimum Steps Required To Accel
65-66	65	22	VNMI	Indirect For NMI - $EB22
	66	EB		
67	67	00	NMIFLG	NMI In Progress Flag
68	68	00	AUTOFG	Auto Drive Initialization Flag
69	69	0A	SECINC	Sector Increment For Sequential
6A	6A	05	REVCNT	Error Recovery Count
6B-6C	6B	EA	USRJMP	User Jump Table Pointer - $FFEA
	6C	FF		
6D-6E	6D	00	BMPNT	Bit Map Pointer
	6E	00		
6F-74	6F	6F	TEMP: T0	Temporary Work Space
	70	00	T1	
	71	00	T2	
	72	FF	T3	
	73	00	T4	
	74	00		
75-76	75	00	IP	Indirect Pointer Variable
	76	01		
77	77	28	LSNADR	Listen Address: Device # + $20
78	78	48	TLKADR	Talker Address: Device # + $40
79	79	00	LSNACT	Active Listener Flag
7A	7A	00	TLKACT	Active Talker Flag
7B	7B	00	ADRSED	Addressed Flag
7C	7C	00	ATNPND	Attention Pending Flag
7D	7D	00	ATNMOD	In ATN Mode
7E	7E	00	PRGTRK	Last Program Accessed
7F	7F	00	DRVNUM	Current Drive Number
80	80	00	TRACK	Current Track
81	81	00	SECTOR	Current Sector
82	82	04	LINDX	Logical Index
83	83	0F	SA	Current Secondary Address
84	84	6F	ORGSA	Original Secondary Address
85	85	3F	DATA	Temporary Data Byte
86	86	00	R0	Temp Work Area
87	87	00	R1	Temp Work Area
88	88	00	R2	Temp Work Area
89	89	00	R3	Temp Work Area
8A	8A	00	R4	Temp Work Area
8B-8E	8B	00	RESULT	Result Of Multiply/Divide Rtns
	8C	00		
	8D	00		
	8E	00		
8F-93	8F	00	ACCUM	Remainder Of Multiply/Divide Rtns
	90	00		
	91	00		
	92	00		
	93	00		
94-95	94	04	DIRBUF	Pointer To Directory Buffer
	95	02		
96	96	00	ICMD	IEEE Command In: Not Used
97	97	00	MYPA	MY PA Flag: Not Used
98	98	00	CONT	Serial Bit Counter
99-A6	99	00	BUFTAB	Buffer Byte Ptrs: Buffer #0 Low
	9A	03		: Buffer #0 High
	9B	00		: Buffer #1 Low
	9C	04		: Buffer #0 High
	9D	00		: Buffer #2 Low
	9E	05		: Buffer #0 High
	9F	00		: Buffer #3 Low
	A0	06		: Buffer #0 High
	A1	00		: Buffer #4 Low
	A2	07		: Buffer #0 High
	A3	00		: CMD Buffer Low
	A4	02		High
	A5	D6		: Error Buff Low
	A6	02		High
A7-AD	A7	FF	BUF0	Inactive Flags For Buffers
	A8	FF		
	A9	FF		
	AA	FF		
	AB	05		
	AC	06		
	AD	FF		
AE-B4	AE	FF	BUF1	Active Flags For Buffers
	AF	FF		
	B0	FF		
	B1	FF		
	B2	FF		
	B3	FF		
	B4	FF		
B5	B5	00	NBKL	Number Of Blocks Low
B5-BA	B5	00	RECL	Low Record # To Find Relative File
	B6	00		
	B7	00		
	B8	00		
	B9	00		
	BA	00		
BB	BB	00	NBKH	Number Of Blocks High
BB-C0	BB	00	RECH	High Record # To Find Relative File
	BC	00		
	BD	00		
	BE	00		
	BF	00		
	C0	00		
C1-C6	C1	00	NR	Next Record Table
	C2	00		
	C3	00		
	C4	00		
	C5	00		
	C6	00		
C7-CC	C7	00	RS	Relative Record Size Table
	C8	00		
	C9	00		
	CA	00		
	CB	00		
	CC	00		
CD-D2	CD	FF	SS	Side Sector Table
	CE	FF		
	CF	FF		
	D0	FF		
	D1	FF		
	D2	FF		
D3	D3	00	F1PTR	File Steam 1 Pointer
D4	D4	00	RECPTR	1st Byte Wanted From Relative Record
D5	D5	00	SSNUM	Side Sector Number Of Relative File
D6	D6	00	SSIND	Index Into Side Sector
D7	D7	00	RELPTR	Ptr To 1st Byte Wanted In Rel File
D8-DC	D8	00	ENTSEC	Sector Of Directory Entries
	D9	00		
	DA	00		
	DB	00		
	DC	00		
DD-E1	DD	00	ENTIND	Index Of Directory Entries
	DE	00		
	DF	00		
	E0	00		
	E1	00		
E2-E6	E2	00	FILDRV	Default Flag, Drive Number
	E3	00		
	E4	00		
	E5	00		
	E6	00		
E7-EB	E7	00	PATTYP	Pattern, Replace, Closed-Flags, Type
	E8	00		
	E9	00		
	EA	00		
	EB	00		
EC-F1	EC	00	FILTYP	Channel File Type
	ED	00		
	EE	00		
	EF	00		
	F0	00		
	F1	00		
F2-F7	F2	00	CHNRDY	Channel Status
	F3	00		
	F4	00		
	F5	00		
	F6	01		
	F7	88		
F8	F8	80	EOIFLG	Temporary EOI
F9	F9	00	JOBNUM	Current Job Number
FA-FE	FA	00	LRUTBL	Least Recently Used Buffer Table
	FB	01		
	FC	02		
	FD	03		
	FE	06		
FF-100	FF	00	NODRV	No Drive Flag: Drive 0
	100	B8		Drive 1 Not Used

Inside the Commodore 64

Chris Zamara, Technical Editor

What makes it tick?

Most of what you learn about your computer involves software techniques of some kind. This article is for hardware fans, experimenters, or the merely curious; since this is the "Hardware and Peripherals" issue, it's only fair that we give you some insight into life inside the plastic case.

The diagram on your right is a layout of the C64's circuit board; this is what you'd see if you opened up the top of the case. There are slight differences among machines, since Commodore occasionally changes the board design to make production more efficient, but overall they should all be pretty similar to this diagram.

Each major section of the board (and a few not–so–major sections) are labelled in the diagram, and described below. Besides descriptions of the parts, there are also usage tips and a bit of little–known trivia thrown in here and there. This is by no means an exhaustive description of the 64's circuitry – that could take up the entire magazine. But when you read the specs for some exotic new sports car, do you really care what the spark plug gap setting is?

The sections are divided into four main categories: CPU and Memory, I/O, Video, and Power. Let the tour begin.

CPU, Memory, and Control

1) The 6510 CPU

The 6510 CPU (Central Processing Unit, a.k.a Microprocessor) has to get the award for the single most important chip on the board, since it is responsible for executing all program instructions. The other chips are just slaves to the master CPU, which controls them at the will of the software, either from the ROM (operating system) or RAM (user). The basic capabilities of the CPU are memory load and store commands, arithmetic and bit–manipulation functions, and two kinds of interrupts. Additionally, the chip has an I/O port built in (accessed with locations 0 and 1) which controls memory bank selection and the cassette unit.

2) The PAL Chip

The PAL or PLA (Programmable Logic Array) is one of the great mysteries of the 64, since it is a custom device unique to this computer – much like a ROM. The PAL chip replaces many discrete gates, multiplexers, etc. and is used to supply the logic required to manage the 64's complex memory architecture. Consider the fact that either RAM, ROM, or I/O may exist in the same address space depending on the whims of the VIC–II video chip, the memory refreshing circuitry, the cartridge on the expansion port, and the programmer himself. All the required logic to handle every possible combination of events lives in the PAL chip, which works like a ROM in that it can be custom–programmed for a given application. PALs, though, are faster than ROMs, and can be used for hardware applications such as these (they're smaller, too). Incidentally, PALs are also more difficult to duplicate or copy than ROMs, and the PAL in the 64 is one factor which makes it difficult to create 64 "clones" – a problem which plagued the Apple II.

3) 8K BASIC ROM

This is the ROM which occupies adresses $A000 to $BFFF (hex). It contains most of the BASIC interpreter; specifically, those parts which are common to all machines. No machine–specific code is in this ROM, so that it can (theoretically) be used unchanged in future Commodore machines, or can survive hardware additions, such as IEEE or 80–column cards.

4) 8K Kernal ROM

The Kernal ROM resides in locations $E000 to $FFFF, and contains – you guessed it – all the machine–specific code. Any routines which deal directly with the video, sound, or I/O chips are found in this ROM. LOAD, SAVE, and VERIFY functions from BASIC are also here. Also, probably due to lack of space in the BASIC ROM, the SYS, COS, SIN, and TAN routines are here. Much of the space is taken up by screen, keyboard, RS–232, and serial port handling routines.

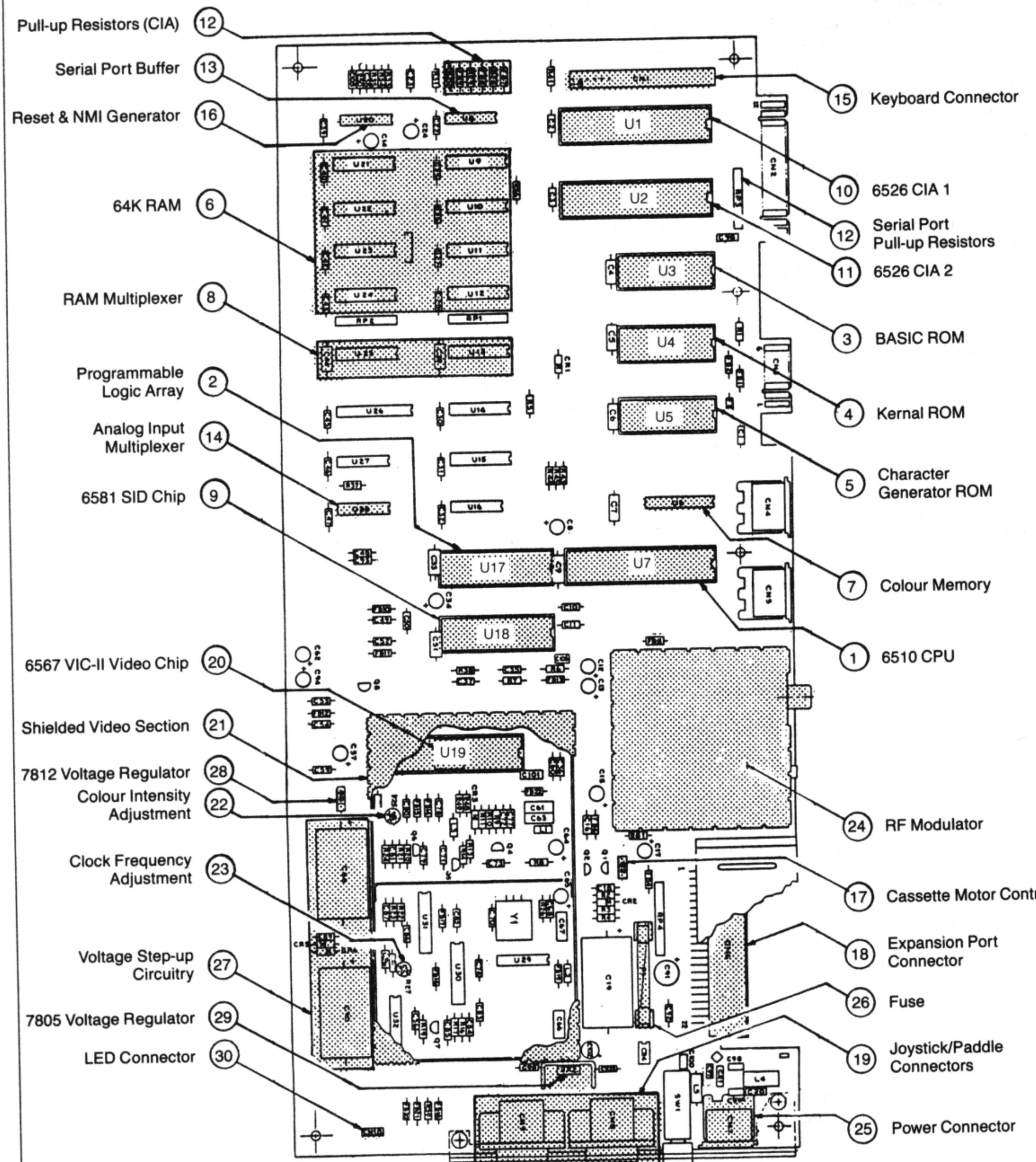
5) 4K Character Generator ROM

This ROM is used by the VIC–II video chip to supply the matrix data required to form all 512 characters: upper and lowercase letters, graphics symbols, and all of the above in reverse–field. The information describing the shape of each displayable character occupies 8 bytes in the character generator ROM. The address space occupied by the ROM is the same as that of the I/O: $D000 to $DFFF. The chip can be read by a user program, by clearing bit 2 in location 1 to "switch out" the I/O. When doing this, care must be taken to stop accessing the usual I/O locations; the normal IRQ handler must be disabled.

6) 64K Dynamic RAM Chips

This is the 64K of RAM from which the 64 gets it's name. Each chip stores 64K *bits*, so eight chips are required to provide complete bytes of data. An interesting usage characteristic of these RAM chips is the way that 64K of memory is addressed. Rather than providing 16 pins on each chip and connecting all 16 address lines directly (all 16 lines are required to access 64K), there are just 8 address lines, and two select lines. The low and high 8 address bits are placed on the chip's address lines alternately, and the select lines are used to tell the chip which are which.

The RAM is *dynamic*, meaning that it only stores it's information temporarily and must be continually "refreshed" so that it doesn't forget. The refreshing is done automatically by the VIC–II video chip, about every 2 milliseconds.



Commodore 64 Circuit Board Layout

7) 1K Colour Memory Nybbles

Colour memory is stored in this 1K by 4 bit chip, which resides from $D800 to $DBFF. That gives 1024 bytes whose lowest four bits (nybble) can be modified. Since the VIC chip supports 16 colours, only four bits are required to represent all possible colours. Each of the 1000 characters on the screen (25 rows by 40 columns) is represented by a nybble in the colour memory chip. The 24 nybbles left over are wasted – don't forget about them if you want to hide a password or something somewhere in memory (sorry if I blew anyone's security scheme).

8) Multiplexers for RAM

The multiplexers are required because of the way the 64K RAM chips are addressed: see item 6 above. These chips translate 16 data lines into 8 data lines and two select lines.

I/O

9) The 6581 SID Sound Interface Device

One of the big, important chips in the system, the SID is the highly–acclaimed "synthesizer chip". I won't get into all of this chip's details; that's the topic of quite a few articles. A brief summary of its sound generation capabilities: three voices, programmable waveform, frequency, attack/decay/sustain/release, pulse–width, high–pass band–pass and low–pass filters, and master volume (quite a mouthful!).

But the SID does more than just sound generation. It also contains two analog to digital converters, which are used for the game paddles. There is a register which indicates the current output of voice 3, and can be used to generate random numbers. And there is an external audio input, which is mixed with the output of the chip and can be controlled with the built in filters. The SID's registers are accessed from locations $D400 to $D41C.

10) 6526 CIA1 Complex Interface Adapter

The 6526 CIA (no relation to the agency of the same name) is an I/O chip containing lots of goodies. It's got 16 I/O lines, two linkable timers, a 24–hour time of day clock with programmable alarm, and an 8 bit shift register for serial I/O. Now, unfortunately, not all of these goodies are effectively used by the Kernal. For example, the 'TI' reserved variable is kept in time (not very well) by software, and the time of day clock in the 6526 isn't even used. The serial and RS–232 routines don't use the shift register, which would allow faster data transfer rates. But much of this CIA is used by the system, in the following ways.

CIA1 is accessed from locations $DC00 to $DC0F in memory. The I/O lines are used for the dual functions of keyboard and joystick reading. Port A, at $DC00, is used as the keyboard matrix row select, and reads joystick 2 or the paddle fire buttons. Bits 6 and 7 of this port are also used to select which analog inputs (paddle set) are read from the SID chip. Port B at $DC01 reads the keyboard matrix column and joystick 1.

The two timers in this chip are used together as a single 16–bit timer. When the timer counts down to its programmed value (set to 1/60 of a second by the operating system), it generates an IRQ (Interrupt ReQuest). The countdown value for the timers can be changed simply by storing new values in the timer locations at $DC04 and $DC05 (low, high). POKEing about here will change the frequency at which the IRQs occur, and can produce some interesting results. Also, the timer can be turned off by clearing bit 0 at $DC0E; an easy way to disable the interrupts – just reset the bit to re–enable.

A readily available source of information on the 6526 itself (not necessarily on the way it is implemented in the 64) is the Commodore 64 Programmer's reference manual. You can also find specs in the Appendices on the 6510 (CPU), 6567 (VIC–II), and 6581 (SID).

11) 6526 CIA2

The second 6526 is used primarily for RS–232 and serial communications, and to bank–select the VIC chip. It's registers lie at $DD00 in the C64 memory map.

Bits 2 through 7 of port A are used for various serial and RS–232 I/O lines (reference: Transactor Vol. 4 Issue 5, pg. 47). The first two bits on this port select which 16K bank the VIC–II video chip addresses. These lines are normally both high, selecting the lowest bank to allow screen memory to reside at $0400. The I/O lines of port B are used as various RS–232 control lines.

The timers in this chip are used by the RS–232 routines and generate an NMI (Non Maskable Interrupt) when they count down. If you're not using RS–232, you can use the timers to generate NMIs for your own routines. This is handy when you need two levels of interrupts, since an NMI overrides an IRQ.

12) R28–R30 and RP3, Pull–up Resistors

R28, R29, and R30 are used to "pull up" some lines on the 6526 which have no internal pull–up resistors (all but the parallel port lines). The resistor pack RP3 is used for the serial port. Pull–up resistors on communications lines reduce noise, but also draw current. That leads to the next component, the serial port buffer.

13) Serial Port Buffer

This buffer chip gives the serial lines more drive current.

14) Analog Input Multiplexer

When game paddles are used, they plug into the joystick ports, a set of two into each port. In the SID chip, there are two registers (at 54297 and 54298) from which the values of analog inputs 'x' and 'y' can be read. Since there are four paddles and only two analog inputs, one or the other must be selected, or "multiplexed" into one. That is the function of this chip. It has two analog inputs and four outputs. One of the two input sets are routed to the outputs, depending on the state of two digital input lines. These select lines are connected to bits 6 and 7 of port A of CIA1 (see 10 above). Before a given paddle set can be read, these bits must be set accordingly:

CIA1, Port A		Paddle set read from
Bit 6	Bit 7	SID Analog inputs
0	0	None
0	1	Joystick port 2
1	0	Joystick port 1
1	1	Both

The reference manual advises giving the lines about half a millisecond to settle after selecting a paddle set before reading the analog value. Instead of four paddles, two analog joysticks may be connected and read in the same way.

15) Keyboard Connector

This connector allows the keyboard to be easily unplugged from the PC board and connects the keyboard matrix to the I/O lines in CIA1. It also contains the line from the "RESTORE" key, which is not part of the keyboard matrix.

16) Reset and NMI (Restore Key) Generator

This IC is responsible for generating the reset pulse to the CPU when the computer's power is first turned on.

It's other function is to generate an NMI (Non Maskable Interrupt) when the RESTORE key on the keyboard is struck. I say struck, and not pressed, because there's an ingenious bit of hardware here to prevent accidental system restores. The effect may vary slightly from machine to machine, but try this: hold down the STOP key as you would before RESTOREing, and then press down on the RESTORE key, ever so slowly and gently. Even when the key is totally depressed, a RESTORE won't occur. This is possible because of the way that the keys on the keyboard "close". Rather than make an abrupt closure at some point of depression, they gradually become more and more conductive as they are pressed down, until the resistance is low enough to bring the 6526's input lines low. That makes it possible for a hardware circuit to only function when the key is pressed down relativley quickly; a capacitor takes care of that.

17) Cassette Motor Control Transistor

This transistor is controlled by bit 5 of the 6510's I/O port (accessed from location 1), and supplies the current required to run the motor in the external cassette unit. The output of this transistor is 6 Volts, and it goes to pin 3 on the cassette port. You can use this to power other devices if you wish, for example a 6 Volt relay. Using a relay in this way would let you do wierd and wonderful things like controlling the lights in your house with software.

18) Expansion Port Connector

This is where the plug–in ROM cartridges go. CPU control lines including the address and data bus come out onto the expansion port. Also available on the port are: The 5 Volt supply, IRQ and NMI interrupt lines, the 8.18 MHz video clock, and the 6510 RESET line. The GAME and EXROM inputs are used by ROM cartridges to switch the ROM into the memory map in place of the usual RAM.

19) Joystick/Paddle Connectors

The joystick connectors are designed to be used with joysticks or paddles, but since they have five I/O lines, two analog inputs, and power available on them, they could be used for other appplications as well. An analog joystick, for example, could be connected to the analog inputs. By using both joystick ports, you would have ten I/O lines available, to which you could connect a keyboard matrix for an external numeric keypad.

Video

20)VIC–II Video Chip

Here's another one of the heavyweight chips that made the 64 famous. Unlike the 6510 and SID, the VIC–II stays out of the limelight by hiding under a large metal lid. The VIC–II (successor to the chip used in the VIC–20) handles all the processing when it comes to the screen. It supports text and bit–mapped screen modes, in normal or multi–colour. The VIC–II is also responsible for generating the "sprites" (or MOBs – Movable Object Blocks), and can be programmed to generate an interrupt when a sprite "collides" with text, or with another sprite.

The VIC–II accesses memory directly, and screen memory can be set up anywhere, within a 16k boundary. Since the VIC–II can only address up to 16k of memory at a time, the upper two bits of the address are formed by bits 0 and 1 of port A in CIA2 (see 11 above).

The VIC–II's various control registers are accessed from $D000 to $D02E in memory. Most of these are used to control the 8 sprites, but there are a few assigned to more esoteric functions. The registers at $D013 and $D014 give the current (x,y) screen location of the light pen, if one is connected. The raster register (at $D012) allows you to select any screen line, and have the VIC–II chip generate an interrupt when the raster beam reaches that line. The source of an interrupt can be examined from register $D019 (there are four sources), and any of these sources can be masked via location $D020. The four interrupt sources are: The raster compare register, a sprite collision, the timer in CIA1, and the light pen input.

21) Shielded Video Section

This section appears as a big metal shield which dominates the right side of the PC board. The metal shield is there to prevent the high frequencies which are present from interfering with radio or television reception. The shield also serves as a heat sink for the VIC–II chip, so you shouldn't run the computer with the shield removed.

Under the lid, the circuitry is divided into two parts. On the left is the VIC–II video chip and a few support components. On the right half is the clock circuitry for the system (1.02 MHz), and for the video (8.18 MHz).

The clock circuitry provides the basic time units, or cycles, on which everything operates. The main system clock of 1.02 MHz keeps the CPU, memory, and other devices synchronized. The length of time taken for the CPU to execute an instruction is measured in clock cycles, since the CPU can do one simple operation (such as a memory read or write) every cycle.

The video clock is 8.18 MHz, the rate at which the indivual picture dots in the video signal are shifted out. It is interesting to note that 4.5 MHz is the highest frequency at which a television can resolve a single dot, which is why two dots must be placed side by side before they become visible. That's why the character set in the 64 is different from the PET's: the 64 must always have two dots side by side, while the PET's higher resolution video monitor can resolve a single dot. (40 column PETs use an 8 MHz video clock, and 80 column machines use 16 MHz.)

22) Colour Intensity Control

This small potentiometer is located in the video section (under the lid) and controls the colour saturation on the screen. In other words, turning the control down makes the picture look more like black and white.

23) Clock Frequency Adjustment

This adjustment fine–tunes the main system clock frequency. But before you try to turn it to maximum to soup up your 64, stop: it only changes the frequency very slightly, and it makes a most profound influence on the screen colours, since it affects the video clock as well. In fact, it's intended purpose is as a tint control, so that the colour clock can be adjusted to make green look green, blue look blue, etc. This is usually set properly before the 64 leaves the factory, so I would advise against playing with it, unless you don't mind the characters in your favorite video game sporting green faces!

24) RF Modulator

The RF (Radio Frequency) modulator's purpose in life is to mix the video signal from the VIC–II chip with a high–frequency carrier so that it can be sent to a television. In some computers, like the VIC–20, the RF modulator is external to the computer. Commodore got it right this time and hid it in the machine where it belongs. The RF modulator is actually a collection of components, like little coils and such. The whole mess is shielded under the metal lid you see to protect the world from spurious RF transmissions.

Power

25) Power Supply Connector

This connector goes to the external power supply. Two power inputs from the supply are fed to the pins of this connector: a 5 Volt DC regulated supply, and a 9 Volt AC supply.

The 5 Volt supply is used to power all the digital ICs on the board except the VIC–II video chip. The 9 Volt AC supply is converted to 12 and 5 Volt DC regulated voltages and used to power the VIC–II chip and clock circuitry (the VIC–II chip requires both 5 and 12 Volt power inputs).

The 9 Volt AC supply has another interesting use: it is the source of the 60 Hz signal used by the time–of–day (TOD) clock in the 6526 CIAs. The 9 Volts AC is fed to a NAND gate inside U27, where it is converted to a 5 Volt square wave. This digital signal goes to the TOD pin on each 6526, where it serves to keeps time for the TOD clock.

Some related trivia: on the SX–64, a switching power supply system is used, and there's no 9 Volt AC supply available. To provide a 60 Hz signal for the TOD clocks, an additional oscillator circuit, including another crystal, exists on the board.

26) Fuse

The fuse is for the 9 Volt AC line only, which means that you won't blow it by shorting out the 5 Volts available on the user port – you'll just shut down the regulator in the power supply temporarily. If you short out the 9 Volt supply on the user port, however, the fuse will blow. The power indicator LED DOES NOT GO OUT when the fuse blows, since it indicates power on the main 5 VDC line. So if your computer dies, suspect the fuse even if the LED is still on.

27) Voltage Step–Up Circuitry

These components are necessary for the process of converting the 9 Volts AC from the power supply into 12 Volts DC for the VIC–II chip. A few diodes and capacitors create a voltage doubler circuit, rectifier, and filter; this converts the 9 Volt AC supply into the 16 Volts filtered DC which is fed to the 7812 voltage regulator.

28) 7812 12 Volt Regulator

After the 9 Volts AC is converted to 16 Volts DC, it is fed to this voltage regulator, which gives a regulated 12 Volt output. The 12 Volts is required for the video circuitry, and connects to the VIC–II's 12 volt input.

29) 7805 5 Volt Regulator

The 9 Volt supply is rectified and filtered, then fed to this IC which produces a 5 volt regulated output. The resulting 5 Volt supply feeds the VIC–II video chip and the clock circuitry under the steel shield.

The 7805 should have a heat sink mounted on it, but some of the earliest 64s came without one. The result was that these machines had problems coping with warm temperatures, and the 7805 would sometimes overheat and shut itself off. No damage to the computer, but imagine the computer dying after you've just typed in a 2000 line program! Apparently, Commodore realized that it wasn't worth saving the 2 cents or whatever a heat sink costs, and all 64s now have it. If your 64 has overheating problems, see if it has the heat sink. If not, you can put one on yourself. Alternatively, the 7805 can be coated with heat–sink compund and pushed up against the steel RF shield to help sink the heat.

30) Power Indicator LED Connector

This connector attaches to the long pair of wires which lead to the power indicator LED. The wire must be unplugged from the connector when removing the upper half of the case.

As mentioned before, the LED does not go out when the fuse blows. It's purpose is to indicate that the regulated 5 Volts from the power supply is alive, since this supply would automatically shut down if overloaded.

Where's the Plugs?

Considering the complexity of the main chips in the 64 and the computer's relatively low cost, there's a surprising amount of stuff in there! Of course, there are many components and circuits not mentioned above, but they are not important enough to go into detail about. Unless, of course, you wish to modify or service the machine yourself; for if you end up buying that sports car, you probably *will* want to know the spark plug gap size.

Thanks goes to "Hardware Corner" Author Domenic DeFrancesco for the technical information which made this article possible, and to Commodore for the circuit board layout diagram.

All About Printers:
What you should know before buying

Chris Zamara, Technical Editor

"Choosing the 'best' printer is like choosing the 'best' car or 'best' pet. . ."

A printer is one of those peripherals that people either have, or wish that they did. For any kind of serious use, you usually need some kind of permanent printed record of your program's output. Even just for program development, it's a lot more productive to sit down with a listing and a pencil than to stare at 25 lines of code at a time on the screen.

But I don't have to convince you how useful a printer is. The problem is, what kind of printer best suits a Commodore owner's needs? Choosing the "best" printer is like choosing the "best" car or "best" pet: it depends on your needs, expectations, budget, and dozens of other personal factors. The best decision is the result of careful compromise, and can only come from knowledge on the subject. That is the purpose of this article: to arm you with information as you enter the ever–growing world of microcomputer printers.

Interfacing and Compatibility

Before going into printers in general, this section must be presented; a general article on printers is no good to a Commodore owner unless he first knows about compatibility with Commodore equipment.

There is some confusion in buying a printer for a Commodore 64 or VIC. You can get a printer which goes on the serial port (like the Commodore printers), an RS–232 printer, Centronics–type parallel printer (which can be directly connected and used with special software), or an IEEE parallel printer with an interface card. Out of all these schemes, only a Commodore printer is directly compatible, since it will connect without an interface, print all special graphics symbols, and produce program listings which look just as they would on the screen. That doesn't mean that a Commodore printer is the only way to go, however. There are a great variety of printers out there that will work very well with a Commodore system. But be careful about your choice of printer and interface, or you may wind up with a printer that you won't be able to use for program listings, or one that will give you lowercase when you want upper, and vice–versa.

An interface lets you connect printers using communication protocols other than those used by Commodore. The most common protocols used with microcomputer printers are outlined below.

IEEE–488

(Usually called "IEEE", pronounced "eye triple ee"). Connects directly to PET, or can be used on C64 or VIC with appropriate interface card. Hence referred to as "IEEE".

Centronics–type parallel

The most commonly used protocol among parallel printers: can be used on 64 or VIC with an interface (eg. cardco), or plugged directly into the user port on any machine and used with special software. Can be used on PET's IEEE bus with an interface which may also convert from CBM ASCII to real ASCII. Referred to as just "parallel" from now on.

C64 serial

Used on Commodore printers: plugs into serial port on C64 or VIC, and can be connected along with disk drives and other peripherals.

RS–232

RS–232 is the standard serial protocol, used by many printers. An RS–232 printer will plug into a C64 or VIC with Commodore's RS–232 interface (this interface may not be necessary with some printers, just a cable with appropriate connectors). C64 and VIC have built–in software to communicate with RS–232 devices – just open a file to device number 2. An RS–232 printer can also be used on PET's IEEE bus, with an appropriate interface.

Here is the danger when buying a non–Commodore printer: some of the Commodore control characters will be misinterpreted. For example, the "home" character is 19 in CBM ASCII, but to many printers, a 19 selects "offline" mode. That means that one of your everyday garden–variety programs could halt the printer while being listed, the culprit being a 'PRINT "[S]"' command embedded somewhere in the program.

Fortunately, there are some very smart interfaces that get around the problem quite elegantly. For example, the CARD? interface from cardco inc. allows you to connect a C64 or VIC to a parallel printer through the serial port (the

same port the disk drive is connected to). The interface knows all about the Commodore control characters, and has a special "listing mode" to deal with them. In listing mode, a home character appears on the printer as "{HM}", a screen clear as "{SC}"; all control symbols including cursor controls and colour commands are coded similarly. You still won't be able to see the special Commodore graphics symbols as they appear on the screen with anything but a Commodore printer. That may be a factor If you're running a lot of graphic–oriented software.

Of course, there is also the compatibility problem with Commodore ASCII and real ASCII: codes for upper and lower case are switched. The interface handles that problem as well, and usually can be selected between PET and real ASCII by means of switches, or by software commands. So don't balk at buying a non–Commodore printer, just make sure that you can get an appropriate interface for the printer you buy, and include the cost of the interface when comparing prices.

An RS–232 printer can be plugged into the RS–232 port with only a simple interface, and it will work, at least from a hardware standpoint. But the compatibility problems outlined above will persist, and may require you to write custom software to get proper listings or convert to real ASCII.

Connecting a parallel printer directly to the user port (same connector as the RS–232 port) will certainly need special software, which is readily available. But word processors and other commercial packages probably won't work with the printer at all; you save the cost of an interface, but you'll mostly be able to use the printer only with your own programs.

An IEEE printer is the best choice for use with any PET–series machine, since it will directly plug in, with the aid of an IEEE cable. Some printers, such as the Epson MX80, can be ordered with optional IEEE input instead of its usual parallel interface. If you have an IEEE card on you C64, you can also use an IEEE printer.

Now, all this talk about incompatibility with commodore control characters won't concern you if you plan on using the printer only with commercial software packages. Most wordprocessors, file–managers, etc. know how to handle different printers, and there are programs available which will let you LIST to a non–Commodore printer. But this brings up another important point: Make sure that any software you buy will work with your particular printer/interface combination! You might walk out of a computer store with an expensive printer and wordprocessor, only to find that you can't get proper printouts. If you're not sure what's compatible with what, you're best off buying from somewhere that has competent salespeople, even if their prices are higher than the local department store.

Of course, there's no "best" setup as far as interfacing goes. The type of communication protocol used on a printer should only influence your purchasing decision if it's going to present a problem as far as interfacing goes, or if the cost of the required interface is prohibitive.

Print Technologies

There are quite a few print technologies in use today. The ones most used for microcomputers are:

Daisy wheel
Dot matrix (impact)
Ink–jet
Thermal
Thermal Transfer
laser

Most home–oriented printers are of the daisy wheel or dot–matrix variety, but there have been great advances in the field of ink–jet, thermal transfer, and laser printers. The merits and faults of the different categories will be given later, but here's just a brief explanation of how each basically works:

Daisy Wheel

The printer gets its name because the characters are at the ends of the spokes of a wheel. The wheel spins until the desired character is in front of a print hammer, where it is struck to impress the ribbon on the paper. The print quality of the characters is usually comparable to that of a typewriter, and different wheels can be used to provide different typefaces.

Dot Matrix

A dot–matrix printer forms its characters from dots. Each dot is produced by a little print hammer striking the ribbon in front of the paper. The appropriate hammers in a row of 7 to 9, arranged vertically, strike the ribbon simultaneously before the print head moves the hammers to the next position.
Matrix printers are generally faster and less expensive than Daisy Wheels, and can often do graphics as well.

Ink–jet

Ink–jet printers are only recently being used with microcomputers. Hewlett-Packard recently introduced the Think-Jet, a low cost Ink jet printer using disposable ink cartridges. And radio shack makes an ink–jet colour printer for the TRS–80 colour computer. Ink jet printers work by spraying ink directly on to the paper through tiny holes, and have the potential of combining high print speed with good print quality.

Thermal

Thermal printers have been around for a long time, but aren't very popular because they require costly thermal

paper. Thermal printers form characters in the same way as matrix printers, but use little heating elements instead of print hammers to burn marks into the heat-sensitive paper.

Thermal Transfer

This is a new technology, and works on a thermal principal, but with regular paper. The printhead heats up the ink in the special ribbon, and boils it out onto the paper. Thermal transfer printers can be fast, give good quality, and print in colour.

Laser

Well, laser printers can't be considered for home use because the cost is still too high, but they're mentioned here for completeness. A laser printer is like a photocopy machine, but it uses a laser to etch the image to be printed onto a photosensitive drum. Laser printers are very fast (think of how long it takes a photocopier to turn out a page) and are used on mainframe systems as the primary high-speed printer. These large laser printers cost literally millions of dollars, and spew out printed paper as fast as 12 inches per second. Small laser printers for micro use are available for between five to ten thousand dollars, and are excellent for office use.

Cost

Obviously, since you don't want to spend all your beer money for the next 18 1/2 years on a printer, the cost criterion is vitally important. I'm keeping away from quoting actual prices, since they're changing all the time, and there are such differences between Canadian and American prices. Depending on how many features and how much speed you need, you can spend anywhere from a couple of hundred to several thousand dollars. Generally, dot matrix printers are cheaper than daisy wheel, and small thermal printers are often the cheapest.

Letter Quality

Letter quality is usually the main reason that people choose a daisy wheel printer over a dot-matrix one that is faster, quieter, and cheaper. But don't think that you must have a daisy wheel printer simply because you want to print correspondence. Most modern matrix printers give very readable copy, and many have a "correspondence quality" mode which gives better quality by printing more dots, at the expense of speed.

Ink-jet and thermal transfer printers use matrix-formed characters, but give better quality characters than impact dot-matrix printers, since the dots blend together more.

A matrix printer can be used to print letters, but for business use or important documents like job resumes, even good matrix may not be good enough. Even so, if these applications are the exception, rather than the rule to your printing needs, you might want to get a dot-matrix printer anyway and make friends with someone who has a daisy wheel. And if a very short letter has to look just right, you can always print it on dot matrix, then do something totally strange like use a TYPEWRITER to get your final copy.

How do you choose between a daisy wheel printer and a matrix printer for the same price, but 10 times faster? Well, what do you want your printer for? Here's two scenarios.

George is a hard-core hacker. He uses his computer for programming and playing with, as a hobby. He needs a printer for program listings, and to print documentation for programs that he writes or utilities that he uses. He also has a modem and accesses bulletin board systems, and he wants hardcopy of some of the information he gets from them. He Makes intensive use of the graphics capabilities of his computer, and would like to print graphs and pictures on his printer.

Jeff has a small business, and wants to use his computer to print invoices, mailing labels, and letters to suppliers and customers. All of his output is textual, and he wants to give a professional impression. He doesn't do much of his own programming, but uses commercially available business packages.

Clearly, George should get a matrix printer of some type (which includes dot-matrix, thermal, and thermal transfer), and Jeff needs letter quality: he'll probably be best off with a daisy wheel. Most of us are somewhere between George and Jeff, and it's just a matter of setting priorities.

Speed

Expect speeds of 80 to 160 characters per second (cps) from dot-matrix printers. The cheapest (and slowest) daisy wheel printers fly along at 12 cps (sarcasm intended), and the fastest about 22. Hewlett Packard's ink-jet printer is fairly fast: 150 cps. Thermal printers are usually somewhere between dot matrix and daisy wheels in terms of speed.

To give you an idea of what these speeds mean, a full page can be printed in under a minute at 120 cps. If you're constantly printing many invoices or long program listings, speed will be an important consideration. For the occasional single-paged letter, you might not care if you have to wait 10 minutes instead of 1, and you can have good print quality at low cost.

You should be aware that the CPS rating of a printer is by no means an average speed. It is a best-case measurement, taken at the fastest printhead speed. If you were to calculate the time a printer should take to print a page based on the numbers of characters on the page and the speed of the printer, the result you'd get would probably be less than half the time that the printer would actually take. There are other factors affecting speed besides the CPS rating, such as bidirectional printing, explained later.

Physical Capabilities

Feed type

Friction feed printers allow you to use single sheets of ordinary paper, just as in a typewriter. Tractor feed uses pins to guide the paper, and requires the use of fanfold printer paper with holes in the side. Tractor feed is better for long printouts, since the forms are continuous and do not require reloading after every page. Also, tractor feed keeps the paper perfectly straight. Friction feed has the advantage of allowing you to use fancy paper like letterheads, or to print on anything: labels, cheques, envelopes, etc. Many printers give the choice of friction or tractor feed, or have tractor feed as an option. It's best to have both.

Characters per line

Most printers will accept standard size 8 1/2 by 11 paper, which is 9 1/2 inches wide including the sides for the tractor pins. Some Larger printers accept wider paper, for example 13 or even 17 inches. The number of characters you can print across a page is an important consideration, especially for work with charts, tables, and spreadsheets. Most printers will fit 80 characters across on a line, which is fine for letters and such, but some printers offer up to 232 characters per line (cpl). There are inexpensive printers which fit only 32 or 40 cpl, and aren't really good for anything other than programming uses (listings, variable dumps, etc).

Pitch

The pitch is the number of characters per inch. A common pitch for printers is 10 cpi, and different pitches may be selectable by software. Check the pitch capabilities on a printer to see if there is an acceptable range. Daisy wheel printers may have pitches ranging from 10 to 15 cpi, and matrix printers from 6 to 17 cpi. Thermal and inexpensive daisy or dot-matrix printers may have a fixed pitch. It's nice to be able to change pitches within a document for emphasis of certain passages.

Noise Level

Some daisy wheel printers sound like a machine gun, and that means you may not be able to use it at 3:00 AM, just when you really need it. Thermal an ink-jet printers are the quietest, being virtually silent. Matrix printers can be reasonably quiet, but some can stand your hairs on end, sounding like a 200 horsepower dentist's drill. When hearing a printer demonstrated in a store, ask yourself if the same noise level will be acceptable in it's intended environment.

Durability/portability

For home use, you may not need an indestructible printer, but if it's going to find use in an office or classroom, you need something big, heavy and intimidating so that people won't destroy it. It might be worth spending more for something with a bit of extra armour-plating. Alternatively, if you're going to be moving it around a lot, you might want something portable, or at least briefcase-able. Some printers will operate from batteries, so that you can use them with a portable computer.

Print Features

bidirectional/logic seeking

Bidirectional means that the printhead prints a line in both directions – from left to right, and on its return path towards the first column. This speeds things up considerably.

Logic-seeking is another time-saver. Logic seeking means that the printhead only moves to where text is to printed – not necessarily the beginning of the line, and it stops where the text ends at the end of the line. When considering the speed of a printer, take into account whether it's bidirectional and/or logic-seeking (every person with a logic-seeking head should do so).

buffer

Some printers have a built in "buffer" to store text temporarily before it's printed, freeing up the computer for the lengthy print process. With a large enough buffer, you could send a document to the printer and regain control of your computer almost immediately, while the buffer's contents are printed. Printers come with buffer sizes varying from 80 characters (one line) to up to 8K, and sometimes a larger buffer is available as an option.

true descenders

This doesn't apply to daisy wheel printers, and all but some of the cheapest matrix printers have true descenders today. Descenders are the bottom part of some lowercase letters, like the tail in "y". True descenders should extend below the bottom of the line, or else the letter looks quite silly. Most modern matrix printers have solved the problem by using more print hammers, but check a print sample before you buy to make sure those little tails hang down nice and low.

typefaces

With matrix printers, the available typefaces is an important but often overlooked feature. You'll probably want italics, and some special graphics symbols that you can use for drawing boxes, different thicknesses of vertical and horizontal lines, solid blocks, etc. Some printers also have all kinds of greek symbols (pi, omega, mu, etc.) which are useful in mathematical or electronically-oriented documents. Daisy wheel printers have the capability of changing an entire typeface simply by changing the daisy wheel itself. Some daisy wheels have more characters than others – giving you a greater selection without changing wheels.

With any printer other than a daisy wheel, the available typefaces are a function of the software (firmware, actually) inside the printer. Again, see a print sample of all available characters and see if the printer has what you want.

User–definable characters

To extend the available typefaces to suit your own needs, some printers of the matrix variety allow user–definable typefaces. That means that you can design a character set, much as you can in your C64, and use that custom set for future printing. You could use this feature to define Commodore graphics symbols in a non–Commodore printer.

graphics

If you're getting a matrix printer of any type, you might as well have one that can do high resolution graphics printouts. You can then use your printer as a plotter for drawing charts, graphs, or function plots. You can also use a graphics program of some sort to compose posters, which can be printed out and used for advertising or presentations. Of course, it's also neat to have just for fun, and to impress your friends and neighbours.

Printers may have more than one graphics mode, offering varying resolution (more dots across).

colour

Another nifty and wonderful feature, found on a few impact dot–matrix, thermal transfer, and ink–jet printers. Characters may be printed in one of a few colours (with impact dot–matrix, made possible by a multi–colour ribbon as on a typewriter). Combined with graphics, the printer becomes a proficient picture drawing tool – which may or may not be what you want it for.

proportional spacing

This deserves its own catagory, because on a daisy wheel printer this feature will allow you to produce documents of typeset quality (like you're reading here). Proportional spacing acknowledges the fact that characters have different widths, and moves the printhead accordingly. If you want to use a printer to produce perfect looking documents, worthy of appearing in print, consider proportional spacing.

"correspondence quality" print mode

This applies to matrix printers, and gives more fully formed characters by using more dots. Selecting this mode slows down print speed considerably (and eats ribbons faster), but can be used to print the final copy of important documents, while using the regular fast print mode for rough drafts. A good feature to have if you want to use your matrix printer for correspondence. It's known as "enhanced mode" on some printers.

other features

There are many things that a printer can do for you, and you should know what you need and what the printer can do before you buy. A few of these features are: Superscripts and subscripts, boldface, underlining (proper underlining underlines spaces as well as letters), tab stops, and formatting. Commodore printers have exceptionally good formatting capabilities, which makes it easy to print column of figures neatly. Every printer has its own list of unique features, and you'll have to see the printer manual to discover them all.

Operating Cost and Maintenance

With impact type printers, the operating costs (besides electricity of course) come from the continuous use of paper and ribbons. Ribbons come in two flavours: the cartridge type, and the open type. Cartridge ribbons are very easy to replace: just snap the old one out and the new one in. They are also much more expensive than the open typewriter style ribbon. Check the ribbon type when buying; it's a case of preference – personally, I'd rather get my hands dirty once in a while and save the money (about $2.00 for an open typewriter ribbon vs. $16.00 for a cartridge).

Ink–jet printers use replaceable ink cartridges, and are fairly inexpensive to operate. For example, Radio Shack's CGP–220 uses a three colour cartridge which is good for a claimed 3 to 4 million characters. Cost of replacement cartridges is about $13.00 here in Canada. Incidentally, this printer has a parallel interface and could probably be used on a Commodore system (cost is around $900 Canadian).

Cost of paper is the same for all except thermal printers, which use more expensive heat–sensitive paper. You'll need a box of paper for your printer – here's a little aside: When you buy paper for a tractor–feed printer, you have to get 9 1/2 by 11 if you want the sheets to be the standard 8 1/2 by 11 size after the perforated tractor feed edges are removed.

Good Luck

Now that you know a bit about printers, don't get too smug; the information is changing all the time. New features, capabilities, and lower prices are changing printer technology (for the better) to the point where it's hard to keep up. But armed with the information covered here, and maybe some that I've left out, you won't feel lost in the world of the printer – which may be the most important peripheral in your system.

Evolution Of The CPU And Revolutionary Memory Advancements

Howard Rotenberg
Toronto, Ont.

Evolution of The CPU

This should take some of you older folks back to memory lane (no pun intended) and give the younger ones a bit of history. The instruction sets of all the early computers contained fixed point arithmetic instructions, boolean and shift instructions. There was also a few primitive instructions for controlling the instruction sequence. Then came the second generation computers. These had much larger instructions sets and a lot more flexibility and power. Basically these computers fell into two categories: business and scientific computers.

The business data processing computers were typically character oriented and had an enormous capability for manipulating variable length data. These operations were usually character replacements or comparisons for sorting, merging or fixed–point arithmetic. The arithmetic was generally slow on this type of computer since it was usually done serially by character, and often done decimally rather than in binary.

The other computer I spoke about was the scientific computer. Although it had extensive floating–point arithmetic capability, it operated on data of fixed precision. It, unlike the business computer, had very little capability for dealing with variable length strings and did not have decimal arithmetic, although it could be performed through programming at a high cost of computing time.

Toward the end of the second generation, the distinction between business and scientific computing started to fade. Large scientific users performed computations involving sorting, merging and character manipulation of variable length data that was previously classed as business functions. While this was going on, many business applications started to make use of sophisticated forecasting and inventory control programs which relied on operations of scientific computing. The beginning of the 60's saw a new breed of computer. The manufacturers started to combine both kinds of functions into one computer.

A major obstacle in the way of having a single general purpose computer for both business and science was the cost. As a number of instructions in a computer grew, so did the cost of its control unit which in turn affected the cost of the machine to the user. By the time the middle sixties came about, new technologies and new design techniques brought hardware cost down dramatically. It was now possible to construct a general purpose computer to satisfy all users.

As computers moved into the third generation, instructions were now designed for a large variety of functions. The number of elementary instructions increased from a few dozen in the early computers to over a hundred, and in a few instances to over two hundred. Arithmetic instructions only accounted for a small portion of these repertoires. The newer functions included instructions for subroutine entry and exit, environment changing, stasis recording, and memory protection. Even the least expensive mini computer of the early seventies can perform upwards of one hundred different elementary instructions.

By this time the computer designers recognized the need for modularity in the physical construction of computers. It was only natural that the basic building blocks they used were nand/nor logic gates and flip/flops. At that time these were fabricated from discreet resisters, capacitors, and vacuum tubes. Second generation computers used transistors in place of vacuum tubes. There was some simplifications in the structure of the logic gates but basically the designs were similar to those used previously. Registers were relatively expensive during this time so that multiple registers were found mainly in high speed computers whose users were willing to pay a premium for speed.

During the early sixties several developments in device technology lead to revolution changes in the logic devices available. These changes also lead to dramatic decreases in the cost per logic function. The logic devices by this time were fabricated on flat films deposited on a silicon substrate. This new silicon chip contained several active transistors, yet the cost of chip fabrication was approximately equal to the cost of fabricating a single transistor using previous techniques.

This new technology had a major impact on computer design and then on computer architecture. The costs of batch fabricated devices could be held low if many copies of the chips were made. For the portions of a computer such as memories, arithmetic units, and registors, this new technology fit well and adaptation was relatively simple. But for the unstructured logic of the control unit, there were significant problems.

From this we can see that the new technology invalidated the former design criteria of minimizing the number of logic gates in a design and replaced it with the need to maximize the repetative use of a functional chip, or to minimize the number of interconnections between functional partitions of the control unit. (Boy that was a mouthful). This now provided a partial impetus for implementing the control portion of a computer with memory rather than discreet logic to achieve regularity of structure. This in turn contributed to the rise of microprogramming in the middle and late sixties.

Evolution of Memory

Along with the evolutionary changes in the central processing unit, great advances were also made in memory architecture. The latest and most revolutionary advancement in memory I will discuss towards the end of this article. There are three types of memories that had come into popular use as a primary memory of a computer. In the fifties, the primary memories were mainly rotating drums. The late fifties saw the introduction of second generation of computers with magnetic core memories. These computers were substantially faster than their predecessors because of the new random access capability.

That is, any item can be fetched from memory in a fixed time span that is independent of the previous memory reference. Magnetic drums had what we could call an inherent rotational latency. What I mean by this is that if the item to be accessed was not under the read head of a drum at the precise time the access request was issued, then the request could not be granted until the item eventually reached the read head. During this time the computer remained idle until the access occurred.

The third and most popular memory that was introduced was the integrated circuit memory. Like the magnetic core memories, the integrated memories are random access. They were much faster however and with the new techniques available for the making of these

chips, even cheaper. Integrated circuits are the most popular type of memory used these days and pretty well the only kind found in todays micros.

The organization of primary memory today is essentially just as the early designers envisioned it to be. Items have unique physical addresses, and accesses are made by address. The advances that were made in memory architecture fell primarily in two related areas:

1) Memory hierarchies consisting of high speed first level storage together with much larger and slower second level and possibly, third level storage, appear to the program as a single large memory whose speed is nearly equal to first level storage. (wow that was another mouthful). We would tend to see that frequently used items tend to reside in the fastest memory of the hierarchy and automatically drift to the slower parts of the hierarchy when frequency of use diminishes.

The second I feel is a more important advancement.

2) Several independent programs can run in the same computer system simultaneously, each using a completely separate set of memory locations. Memory address translation facilities simplify the problem of allocating memory to the individual programs, and guarantee that one program cannot interfere with another program. In some instances, specified data or program segments are permitted to be shared by two or more programs.

I will not go into any more detail about these topics other than to mention that the first of the two is called virtual memory or cache memory. The second describes a situation known as multiprogramming and address translation facilities are essential for its support.

A Glimpse of The Future

There is a man who is presently a senior research officer in the Electrical Engineering Division of the National Research Council in Ottawa that has come up with an astounding discovery. His name is Alex Szabo and he unbelievably has found a way to pack more information into a computer memory than a human brain could hold in the same amount of space.

As I have mentioned earlier, for years data has been stored on discs, tape and more recently bubble memory. Now this man has added the concept of the crystal memory. The potential of this discovery is overwhelming. His idea was originally conceived in the early 70's.

To briefly remind all how data is stored, it is essentially one of two kinds of signals or states. This is on or off, whether the data is on tape, disk or in other types of memory. For instance. eight bits may be stored in a certain sequence to represent a particular character. Szabo's idea is quite different in that it takes a crystaline material which light normally can't get through, and makes it transparent to certain colours of light in specific spots.

The first material he used was a ruby, since while a ruby could hold back most light but not a thin laser beam. There are a lot of other crystals in this category also that are even better. He had found out that you could pass a thin laser beam through a thin slab of material and penetrate it. The penetration would leave a tiny tube like section transparent exactly the size of the laser beam. This tube was now clear while the rest of the crystal was still impenetrable to light. These beams can be focused down to a thickness of less than a 10,000 of a centimetre. This is the key to this fantastic capability: This man found out that the tiny track that had been left transparent would only pass the same colour of light as the colour of the original laser beam that penetrated it. If the laser was green and you passed a green light over the crystal, you would only see one dot of green light actually passing through. If you passed a red laser beam through the crystal the same effect would be evident. The last thing that Szabo found out was that if you passed both a red and green laser beam through the same spot, it would be transparent to red and green both but still block all other colours. The next amazing thing that should be mentioned here is that, theoretically, lasers could distinguish between 10 million shades. They can be tuned very precisely to these shades of colour also.

The Next Step

Szabo reasoned that he could arrange a pattern of spots on a slab of crystal and shine red laser beams on some of those spots but not on others. He could then arrange tiny light sensors in an identical pattern on another surface underneath. Now if he shone a flood light of the same red colour on top of the crystal the light would get through only the spots that had been penetrated earlier. This would activate the light sensors beneath those spots but leave all the other sensors dark and inert. This arrangement could be used as a memory cell. A computer could get digital on signals from the sensors which got light, and off signals from the one that stayed dark.

This was a very impressive idea for data storage, because laser beams can be made so thin that could be directed at a hundred million different spots, without overlapping, on a slab of crystal only one centimetre square. This would be about the size of a human fingernail. His idea went far beyond this. If you penetrated a crystal by red laser beams in one array of spots, that would store data which could then be read back by simply shinning a red light on the slab at a later time. If the same crystal was then penetrated by green laser beams in another arrangement of spots, it would store a separate hundred million bits of data, which could be read back later by shinning a green light on it. In all practicality probably only one thousand shades of lasers would be used because today's equipment can tell that many apart very quickly. This would mean that a fingernail size slab of crystal could be used to store up to one million bits of data in each of one thousand different laser colours. We are now talking about the storage of one hundred billion bits of data in a fingernail space.

With this principal in mind, in a crystal the size of a hi–fi music album, it would be possible, theoretically to store as many bits of data as there are neurons in the average adult human brain. From this we can start to see the enormous capabilities that could come from this revolutionary type of memory. There are some problems to cope with here, aside from figuring how to fit a hundred million light sensors into a square the size of a fingernail. To use any crystal material this way, it is absolutely necessary to keep it extremely cold. To be a little more precise, the temperature of liquid helium, 269 degrees below zero celsius is good. The hottest allowable temperature would be approximately 253 below.

The reason for these low temperatures is to slow the motion of the molecules in the crystal almost to zero–to keep them frozen stiff. If the molecules were to move as much as they do at higher temperatures, the material would lose its colour stability and the laser holes could no longer control the precise shades they were made for. To keep such a frigid level high power cooling systems would be needed. Unfortunately, at this time that would mean crystal memories would be practical only for big computers that stay in one place. We wouldn't see them for desk top or portable models for a long time.

Conclusion

I hope that this article has given some insight into the architecture and functionality of the CPU and new advancements that are being made in memory. It reminds me of the show you may be familiar with: WHAT WILL THEY THINK OF NEXT.

Modifying the VIC–20 3K RAM Pack For Use With EPROMs

Thomas Henry
Mankato, MN

An easy way to add EPROMs to your VIC!

If you ever open up your 3K RAM Pack for the VIC–20, you'll be in for a pleasant surprise. Besides the 3K of RAM chips, you will find room on the circuit board for two 24 pin EPROM's! In some cases all you will have to do to use these slots is throw in a couple of integrated circuit sockets and away you go! This article explores the workings of the 3K cartridge and shows several ways to employ the extra EPROM space.

All you'll need, besides the 3K RAM pack, is:

two 24 pin IC sockets,
two .01 mfd. disk capacitors,
and a 74LS00 quad NAND gate

(This should cost you less than a cheap lunch at the local greasy spoon.)

But first, a warning. Any time you start fiddling with the hardware of your computer you expose yourself to several dangers. First, you will void any warranty that came with your system. This may or may not bother you, depending on your outlook towards warranties. More importantly, the object of performing any hardware modification is to improve the system, but if you aren't careful, you may end up destroying some valuable equipment! The general rule, then, is if you don't understand what you are doing, don't do it! Hire someone more skilled in electronics to perform the needed changes.

Having possibly scared you, let it quickly be noted that the modifications to the 3K RAM Pack described in this article are actually quite easy to perform and as long as good shop practices are utilized, you should encounter no trouble. Any outstanding tricky points will be mentioned along the way.

With all of the caveats behind us now, let's examine the cartridge and see what can be done with it. Your first move should be to open up the cartridge and spend some time simply looking at it. There is one screw which holds the cartridge together, and this is found in the middle of the back. Remove this screw. Next insert the blade of a small screwdriver into one of the two slots found along the back edge. Carefully push the interlocking tab out of the way and pull the back up slightly. Now repeat with the other slot. If all has gone well, the back of the 3K cartridge can now be lifted off.

Holding the circuit board thus removed, with the edge connector fingers downward, look for the six RAM chips off to the left. Now look for the two empty EPROM locations on the right side. Finally, locate the eight circular "select pads" between the RAM and EPROM area and towards the bottom of the board. Figure One shows how the select pads should appear in a stock 3K cartridge.

What are these pads? Simply put, the select pads allow you to change the decoding of the EPROM chips and furthermore make the use of three different types of chips possible. Each select pad is composed of two semi–circular regions. Some of these pairs (the ones labelled 1, 5, 11 and 12) have a printed circuit board trace connecting the two halves. The other ones are electrically distinct. By cutting the traces or bridging the distinct pairs with solder, it is possible to configure the EPROM decoding for a variety of uses. Let's get more specific.

Select pads 5 and 2 govern how the upper righthand EPROM chip is to be decoded. Since the printed circuit board comes with select pad 5 bridged and select pad 2 broken, the chip will be decoded by Block 5, or in other words by any address between $A000 and $BFFF. By breaking select pad 5 (cut the trace with a razor knife) and making select pad 2 (bridge the two semi–circles with a dot of solder) you can change the decoding so that the chip is now addressed by Block 2 ($4000 to $5FFF). Note that only one of the two pads (5 or 2) should be bridged at a time.

In a similar manner, select pads 1 and 3 govern how the EPROM in the lower righthand side of the board will be addressed. If pad 1 is bridged, then the chip will be addressed by Block 1 ($2000 to $3FFF); if pad 3 is bridged, then Block 3 does the trick ($6000 to $7FFF). Thus by making or breaking select pads 1, 2, 3 or 5 it is possible to locate EPROM in any area between $2000 to $7FFF and $A000 to $BFFF. That's quite a bit of versatility! (Incidentally, addresses $8000 to $9FFF are reserved by the VIC–20 for character sets and I/O).

But there's more to come! Look again at your exposed circuit board or Figure One and you will see four more pads that haven't been mentioned yet. Two of the pads are associated with the number 11 and two with the number 12. Call these 11–upper, 11–lower, 12–upper and 12–lower. By making or breaking various combinations of these select pads it is possible to set the 3K RAM pack up to use any one of three types of EPROM. These are the 2716 (2K), the 2532 (4K) and the 2564 (8K).

Figure Two shows the combinations of pads needed to accomodate each of the three types of EPROM useable with this cartridge. One important point to notice is that the 2732 (also a 4K EPROM) may not be used with this setup. The pinout is just sufficiently different

from the 2532 to cause problems. So, always stick with the 2532 and you won't go wrong.

Perhaps you have the uneasy feeling that you haven't been told the whole story yet, and you're right. If you stop to think about it, since the chips are decoded using the various block select lines of the VIC–20, only whole chunks can be selected at once. In the case of the 2564 EPROM, this is what you would want since it is desired that any one of the 8K addresses be selected. But in the case of a 2K or 4K chip you would clearly be wasting a lot of space. To put it another way, suppose you are using a 4K EPROM located in the Block 5 region ($A000 to $BFFF). All of the addresses from $B000 to $BFFF can't be used since there is no memory there to be accessed. How wasteful!

Since 4K EPROMs are by far the most common type to be found, let's delve more deeply into how to make the best use of this type of chip. In general, the region between $A000 to $BFFF should be devoted to EPROM. Wouldn't it be nice, then, if we could set up the two empty chip slots in the 3K RAM Pack so that a pair of 4K chips could occupy this entire region? In this way, no addressing space is wasted. Let's see how to do it.

Clearly we will need to add a little extra decoding circuitry since the block select lines decode 8K chunks at a time. (The block select lines are derived from A13, A14 and A15, the highest three address lines). To narrow this down to 4K chunks we will need to combine the BLK5 line (Block 5) with A12 (address line 12) in some fashion. Figure Four shows a circuit which will do the trick. We will look at the circuit in greater detail in just a moment, but for now notice how A12 and BLK5 are input to the decoder and chip selects for $A000 through $AFFF and $B000 through $BFFF are output.

Getting Started

Refer to Figure Three now which shows how to modify the select pads to accept this additional decoding. You will need to break all of the pads, except for 11–upper, and then make 12–upper. A12 can be picked up from the plate trhough hole next to select pad 12–lower. Finally, the BLK5 line can be picked up from the leftmost half of select pad 5. The chip select line for the EPROM addressed to $A000 through $AFFF can be picked up from the plate through hole next to select pad 3. This decodes the lower rightmost EPROM to this region. Not shown in Figure Three is the chip select line for the $B000 EPROM. This can be picked at the plate through hole located next to the positive lead of the electrolytic capacitor in the 3K cartridge. The upper righthand EPROM corresponds to the $B000 region then.

This takes care of the theoretical aspects now. We know where to find A12, BLK5, and the two chip select lines (which run to pin 20 of their respective chips). Let's see how to actually make the changes needed to get two 4K EPROM's up and running.

If you haven't already done so, remove the circuit board from the 3K cartridge at this time. Remove the single screw holding the back on. Then insert a small screwdriver into the interlock slots and release each corner to separate the casing.

You will want to start out by cleaning up the IC socket holes as needed. Unfortunately, Commodore fills in the unused circuit board holes with solder; you'll have to clean them all out again. But the task isn't as bad as it seems at first. Using a small soldering iron and a solder sipper, start by cleaning out the holes in the two EPROM areas. Do not apply too much heat to these, since the traces are relatively fragile and may lift up from the board. If you can't get the solder to come out on the first try, move on to another pad and let the first one cool off a bit. Then come back to the first one and try again. If the hole seems too "dry", add a little extra solder, then try sipping the whole amount out again. However you do it, be patient and careful and watch the heat of your iron! This process takes about ten minutes if all goes well.

Next, right above the pin 1 end of each EPROM area are two holes for decoupling capacitors. Clean these out. Later on we will install two .01 mfd. disk capacitors to decouple the EPROM's from the rest of the circuitry.

Sip out the plate through holes needed for the A12, chip select $A000 and chip select $B000 lines. Finally, put a drop of solder on the 12–upper select pad to make the connection, and break the pads at 1, 5 and 12. Use a razor knife to break the connections, but be very careful not to cut any other traces on the board, or worse, your hands! See Figure Five.

We are now ready to start building. Start off by installing the two 24 pin EPROM sockets, watching most assiduously for cold solder joints or bridges. By the way, be sure to use low profile sockets so you will be able to get the cover back on the cartridge when you're done!

Next install the two .01 mfd. disk capacitors (as mentioned above). These capacitors are strung across the +5V supply line and ground and help keep switching noise from being passed through the rest of the system. Figure Six shows one of these new capacitors in place.

The 74LS00 is installed in a slightly exotic fashion, due to the close quarters on the circuit board. It is actually mounted upside down! Using some 5 minute epoxy cement, glue the chip upside down between the two EPROM sockets. Orient the IC so that pin 1 points towards the center of the circuit board. This is a strange way to do things, but really it's hard to suggest a more practical alternative.

Finishing Up

After the epoxy cement has hardened, you may complete the final wiring. No explicit point–to–point instructions are given here. This is for two reasons. First, each experimenter will have his own techniques and methods for wiring up the chip. Secondly, and perhaps more importantly, if you don't have the confidence to do the wiring without detailed step–by–step instructions, then you shouldn't even contemplate this project! As mentioned above, these modifications are fairly straightforward, but the work is fairly detailed and should only be attempted by experimenters with some experience in electronics. There's too much at stake to have a slip of the soldering iron ruin everything!

Procedure Tips

Even though explicit instructions won't be given, here are a few tips to guide you along. Use bare bus wire and 22 guage insulated wire to accomplish the various connections. You can pick up ground for the 74LS00 (pin 7) off of the heavy PC board trace located at the far right side of the EPROM sockets. The +5V line is

just as easily tapped off of a disk capacitor lead right next to pin 14 of the 74LS00. Using Figures Three and Four, complete the wiring of the decoding circuitry.

The decoding circuitry in Figure Four is simple to figure out if you draw up a truth table. Intuitively you can see that the BLK5 line is needed to turn on either the $A000 or the $B000 chip, and it is the A12 line that determines which of the two chips are to be selected. Note that the outputs are active low.

Remember, there's no rush, so take your time and see that the final wiring gets done correctly. After half an hour's work you should be done and ready to stick the circuit board back in its case.

The New Final Product

What have we accomplished? Well, using the 3K RAM Pack, which we already had, and a handful of inexpensive electronic parts, we have set up the cartridge to accept two 4K EPROM's, one decoded to the $A000 region and one to the $B000 region. This is almost like getting something for free, if you stop and think about it; we get some extra memory without even having to get our hands dirty making a circuit board!

Now that you have this memory area available, what are some good uses for it? Well, I used my cartridge to implement a FORTH package on the VIC-20. Other good uses would be to burn an EPROM version of MICROMON for the VIC or any other utilities that you need frequently. Having programs like these available in EPROM is really neat; they're always there and ready to go! No more long tape loads for 4K and 8K utilities!

We've seen in this article, then, how to squeeze the most out of a 3K RAM Pack. As mentioned, by understanding and using the select pads, it is possible to handle three different types of EPROMs and locate them in a variety of different memory locations. Furthermore, with the addition of a few extra components, we saw how to set up the EPROM decoding to accomodate two 4K units with no wasted addresses. Who would have thought that the 3K RAM Pack was capable of so much!

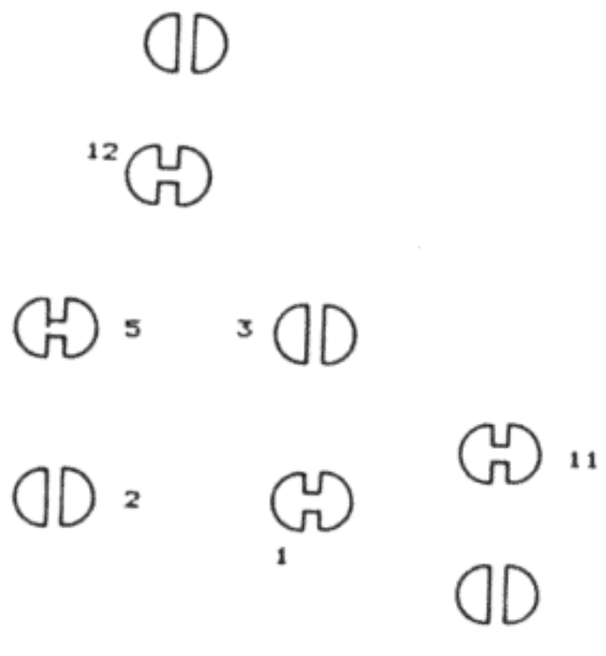


Figure One: Selection Pads in a Stock 3K RAM Pack

Jumper	2716	2532	2564
11-upper	break	make	make
11-lower	make	break	break
12-upper	make	make	break
12-lower	break	break	make

Figure Two: Selection Scheme for Three Types of EPROMS

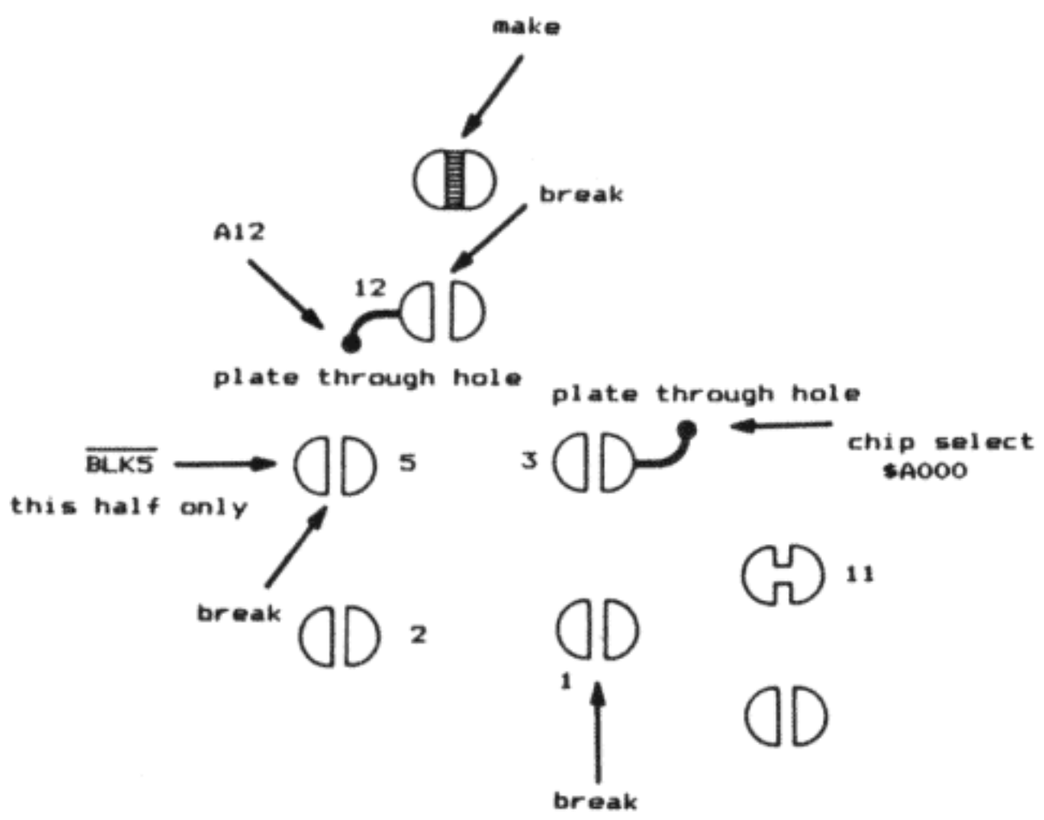


Figure Three: Modified Selection Pads

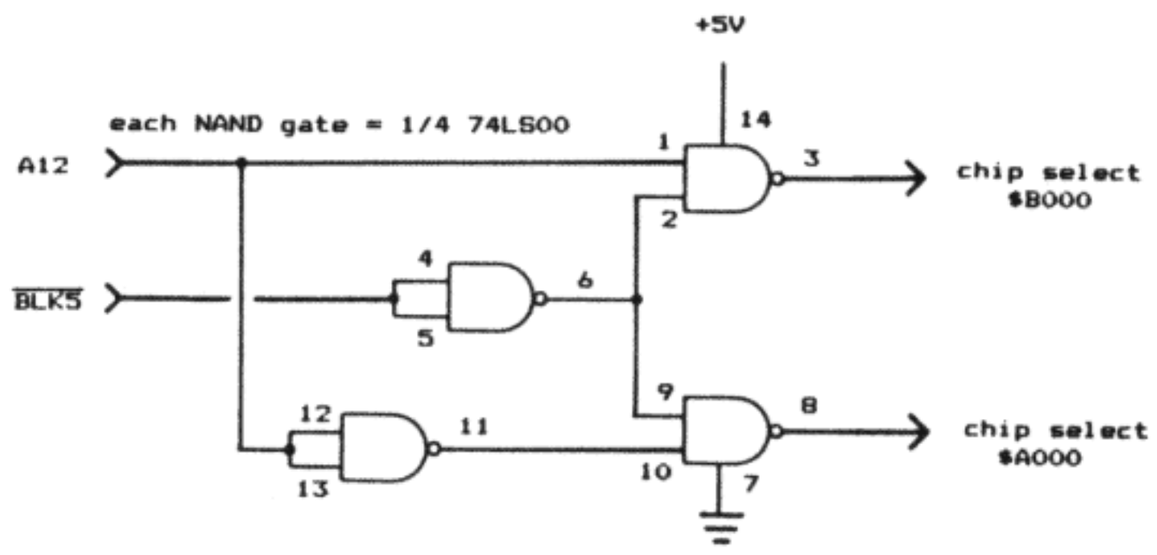


Figure Four: Decoding Circuitry For Two 4K EPROMS

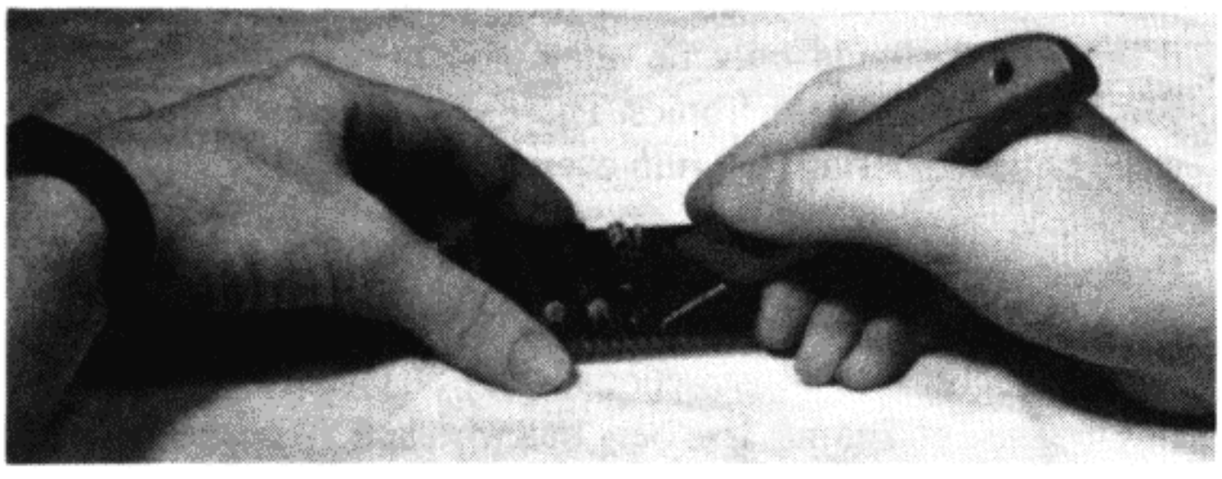


Figure Five: Breaking Connections

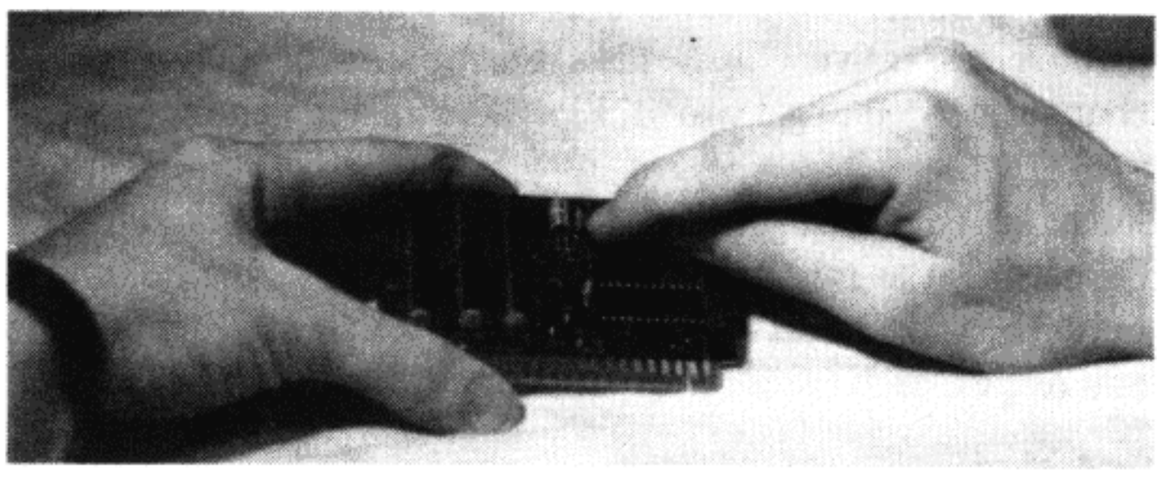


Figure Six: A New Capacitor

Computer Slide Projector Control

Ted Evers
Richmond Hill, Ont.

The following application is one which will interest many. Although the title denotes a specific interface for a slide projector, namely the Kodak Ektagraphic, it can be adapted to control any 120 VAC application that you deem necessary. It is also possible to allow for control of up to 8 totally isolated channels, through program control. Each of these channels can perform whatever duties you assign to them, without requiring assistance in any shape or form from the other channels. As promised, an interface that will interest most.

In brief, the standard interface will allow for forward and reverse motion of a Kodak Ektagraphic Slide Projector under program control via the user port of the computer. In order to help you visualize how this can be achieved, a quick explanation of the port is in order.

The User Port

The ports PA0–PA7 (PET/CBM) or PB0–PB7 (VIC/C64) are programmable input/output lines, driven by POKEs, and read with PEEKs. When programmed as outputs through the Data Direction Register, each line is capable of driving a single TTL input, with approximate current drain of 1.5 ma. When programmed as inputs, each line acts as a single TTL load, with the same current drain expected.

When programmed as outputs through the Data Direction Register, the Output Register will allow for a graduation of voltage levels on the port due to the status of the corresponding bit. If the port is set to 0, the voltage level can be expected to be .5 volt or less. If the bit is turned on, bit=1, then the voltage level will be 2.4 volt or greater. In this way, the port can control virtually any device desired, through the correct interface.

To best understand programmed access to the port, I highly recommend referring back to the Hardware Corner of Volume 5, Issue 1, in the Transactor. The authors cover the subject extensively, and it would be redundant for me to even attempt to cover it once again.

Boolean Logic

Certain techniques are used within the demonstration program shown below that do not fall within the heading of common knowledge. Boolean logic is used within, and will confuse many at first glance. To help alleviate some of this confusion, a quick boolean tour is required.

In order to turn on or off specific bits on the ports without disturbing its current status, boolean logic is best used. You could device a vast formula to calculate the port status with or without your bit, but there is no need. Commodore has incorporated Boolean operators into their version of BASIC, thereby relieving you of the agony.

The operators are AND, OR, and NOT. A quick explanation will be found for each below :

AND

Result includes only that information which is included in both sets.

Set A – 00110101 = Decimal 53
Set B – 11100001 = Decimal 225
Result 00100001 = Decimal 32

Therefore, 53 AND 225 = 32

To find the status of a single bit, (bit 3), mask off other information by ANDing with the decimal value for that bit :

```
PRINT PEEK(59471) AND 8
```

. . .will return 8 if bit is high (on), or 0 if the bit if low (off).

OR

Combines a set so that the result includes all bits "on" which were "on" in either set.

Set A – 00110101 = Decimal 53
Set B – 11100001 = Decimal 225
Result 11110101 = Decimal 245

Therefore, 53 or 225 = 245

To set a bit in a byte, the operator OR should be used:

```
POKE 59471,(PEEK(59471) OR N)
```

. . .where n represents a number in the range of 0–255, will set the bit, or bits, of your choosing.

NOT

Inverts all information within a set.

A – 00110101 = Decimal 53
NOT A – 11001010 = Decimal 202

Therefore, not 53 = 202 (202=255–53)

To clear a specific bit within a byte, AND and NOT are used in tandem:

```
POKE 59471,(PEEK(59471) AND NOT N)
```

. . .where n represents a number in the range of 0–255, will clear (turn off) the bit(s) of your choosing.

When executed directly from BASIC, the NOT command will produce strange results, if taken at face value. From BASIC, NOT 53 = –54, according to Commodore logic. In reality, NOT 53 = 202, (202=255–53). The value of –54 is 256+(–54) which equals 202, the correct result. It appears that the sign bit is flagged, acting

like carry turning on bit 8 (256 value). Therefore, add them together and you get the correct result.

The Program

With the port explained, and boolean logic crystal clear for all, the program, circuit diagram, board design and layout are best shown. Once this has been done, I will explain how the circuit can easily be adapted for any application.

```
100 rem ** ted evers - m.s.s.b. toronto : update sept/84
105 rem ** for use with pet/cbm/c64/vic20
110 print chr$(147)
115 cd$ = chr$(17) + chr$(17) + chr$(17)
      + chr$(17) + chr$(17): rem - cursor down
120 cr$ = chr$(29) + chr$(29) + chr$(29)
      + chr$(29) + chr$(29): rem - cursor right
122 rem  m = data direction reg a, n = output reg a
125 m = 59459:n = 59471 :rem ** pet/cbm
130 rem : m = 56579:n = 56577:rem ** c-64
135 rem : m = 37138:n = 37136:rem ** vic 20
140 poke m,3:poke n,3 :rem set ports 0  +  1 as outputs and
high
145 print mid$(cd$,3)mid$(cr$,4)"    slide projector control"
150 print mid$(cd$,4)mid$(cr$,4)"        demo program"
155 print mid$(cd$,1)mid$(cr$,1)"(f) advances magazine"
160 print mid$(cd$,4)mid$(cr$,1)"(r) reverses magazine"
165 geta$: if a$ = "" then 165
170 if a$ = "f" then gosub 195
175 if a$ = "r" then gosub 245
180 goto 165
185 :
190 rem ** advance magazine **
195 x = peek(n):rem - retain current value
200 x = x and not 1 :rem - turn off bit 0
205 poke n,x :rem - poke it back in
210 k = ti
215 if ti<k + 20 then 215 :rem - wait loop
220 x = x or 1 :rem - turn on bit 0
225 poke n,x
230 return
235 :
240 rem ** reverse magazine **
245 x = peek(n):rem - retain current value
250 x = x and not 2 :rem - turn off bit 1
255 poke n,x :rem - poke it back in
260 k = ti
265 if ti<k + 20 then 265 :rem - wait loop
270 x = x or 2 :rem - turn on bit 1
275 poke n,x
280 return
```

Notes

With the Commodore 64, +5 volts is available through the user port. The PET/CBM was not as well designed. You will be required to tap onto the cassette port to find the power required. The pinouts of all the ports will be required for the correct installation of this interface, and can be found in the Transactor Referance Issue, The Programmers Reference Guide for the C64, and many other books. This I leave for you to locate.

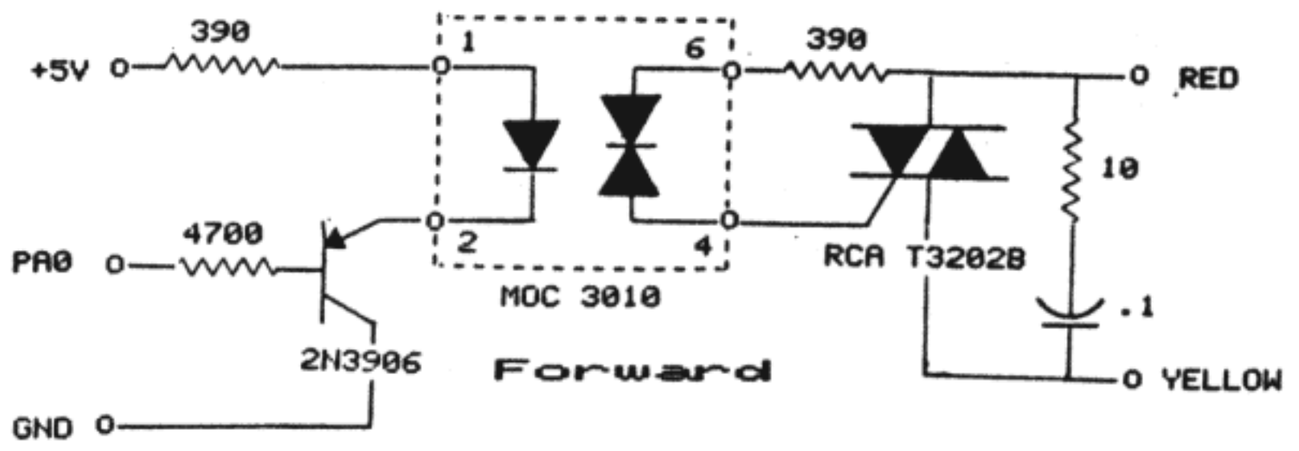
To further increase the power handling capability of the interface, a suitable TRIAC should be used in place of the existing RCA T3202B TRIAC.

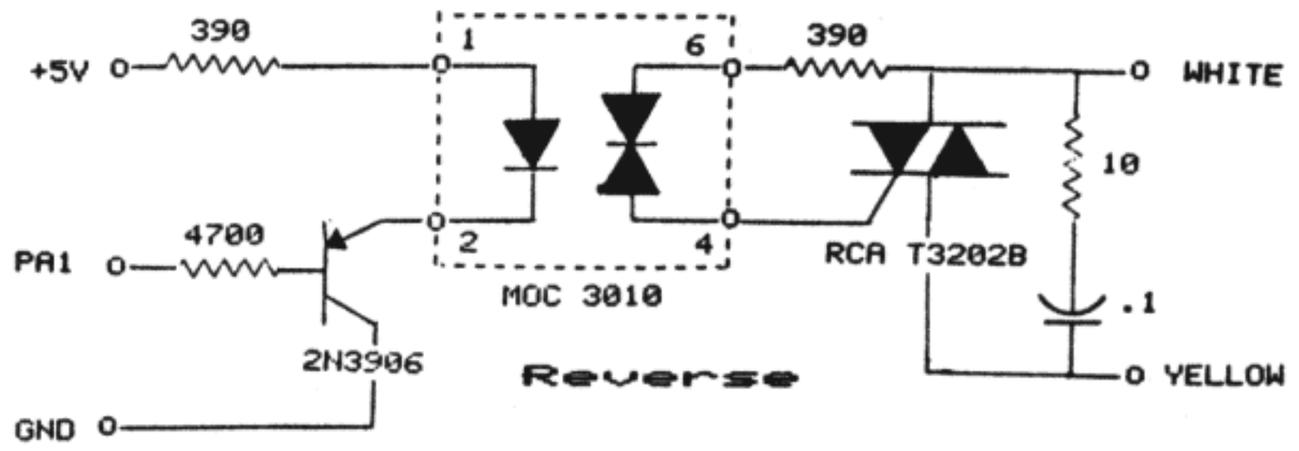
Future Variations

With a few simple changes to this interface, control can be exercised over 8 separate channels, using all 8 user ports lines. For every extra 2 channels required, add an extra circuit board. Each board will allow for control over two ports, therefore, add up to 3 more boards. A simple method to achieve greater results.

To control loads other than one specific slide projector, you will find little effort expended on your part. A diagram (next page) has been prepared to better demonstrate the method.

As can be seen from the diagram, little has been changed. The jumper between the YELLOW line in and the opposite side of the board has been removed, and two separate LOADs have been brought into play. In the diagram it has been shown that 2 sources of AC are required. This doesn't have to be if you bridge the AC lines together. In this way, a single AC line in could control up to eight channels, if you so devised. The choice is yours to make.





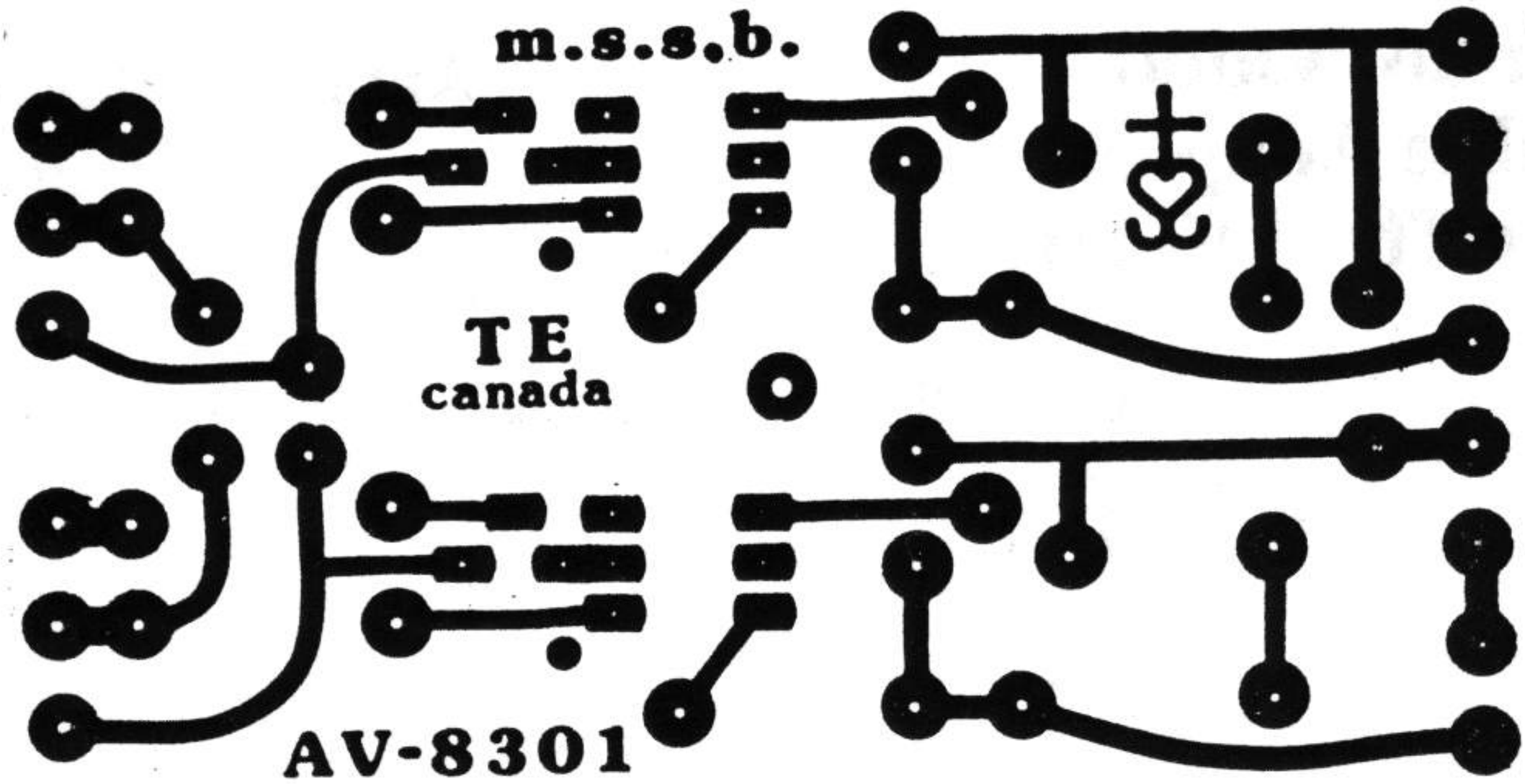
Socket Preh# 71202-030

(Solder View)

Plug Preh# 71418-030

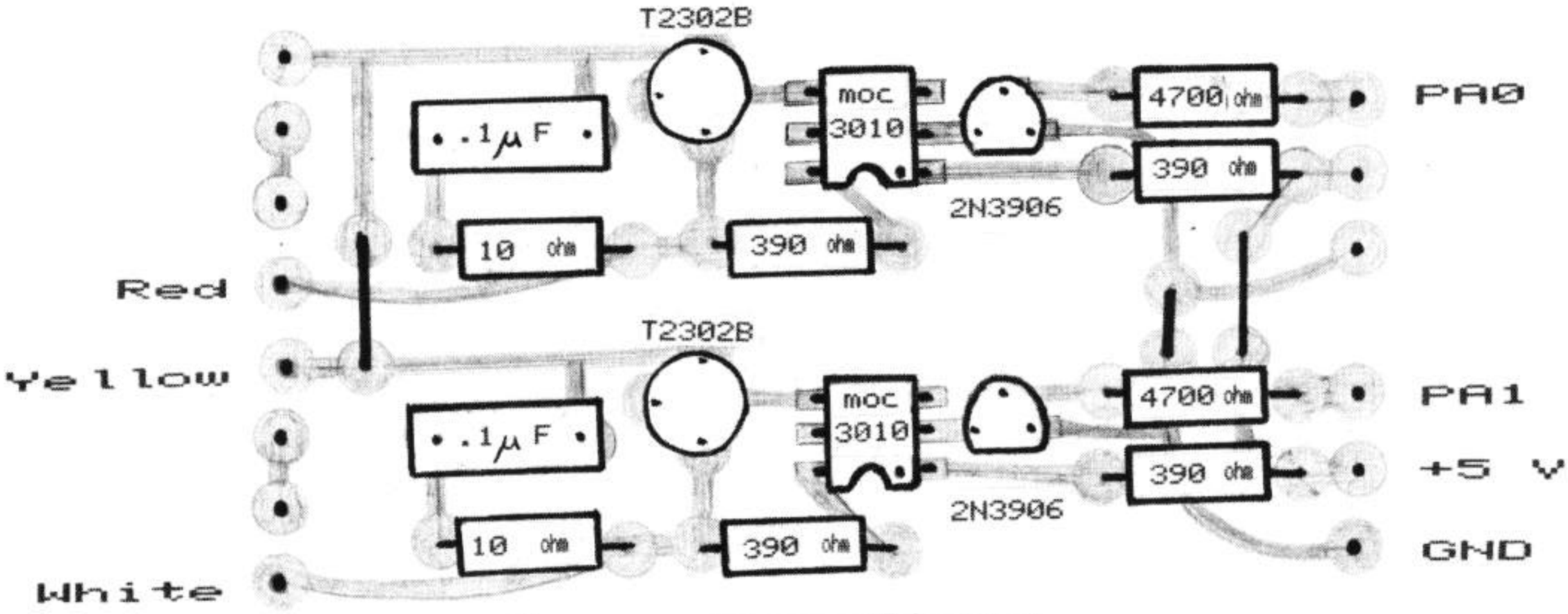
Add This Plug To "Kodak Remote Control" For Patch Cord To Projector

Schematic Diagram

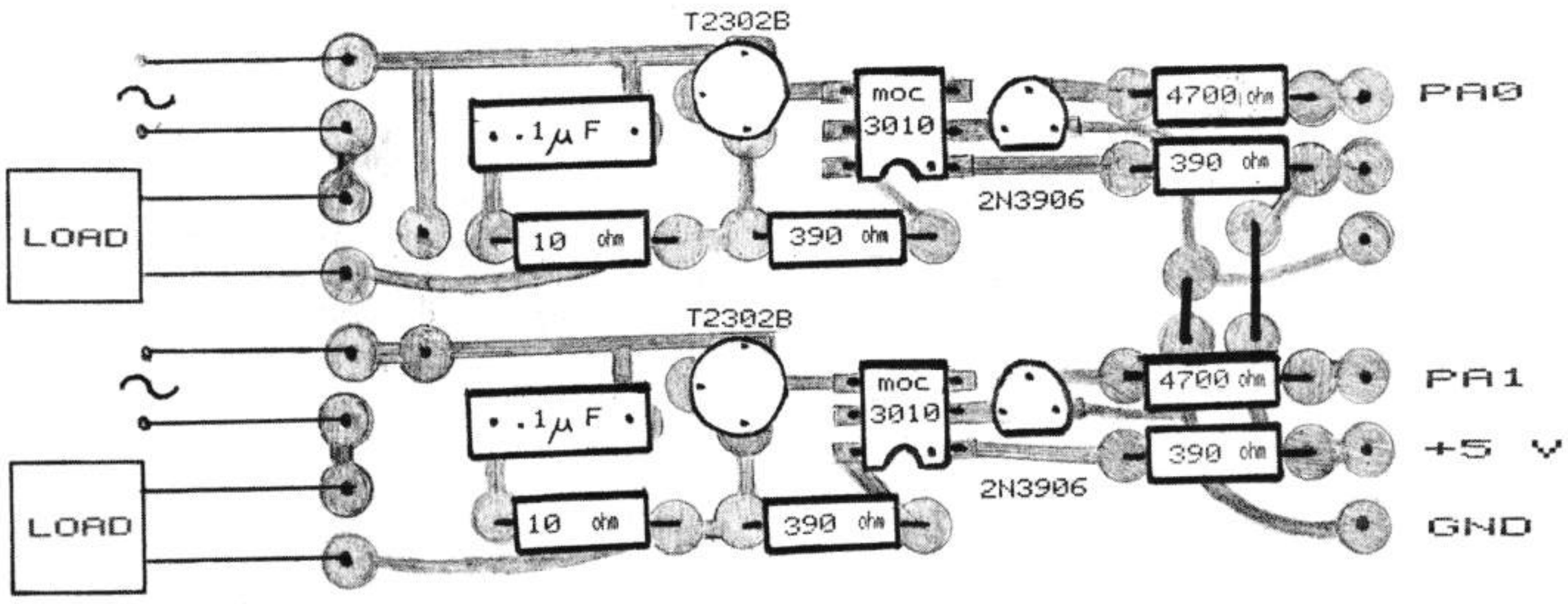


Printed Circuit Board Design

Double size as shown. The board physically measures in at 3 inches in length by 1.75 inches in width.



Component Layout Diagram



Connection of loads and AC lines

Audio/Video Cable Adapter For The VIC 20

Arthur S. Barlaan
Chicago, IL

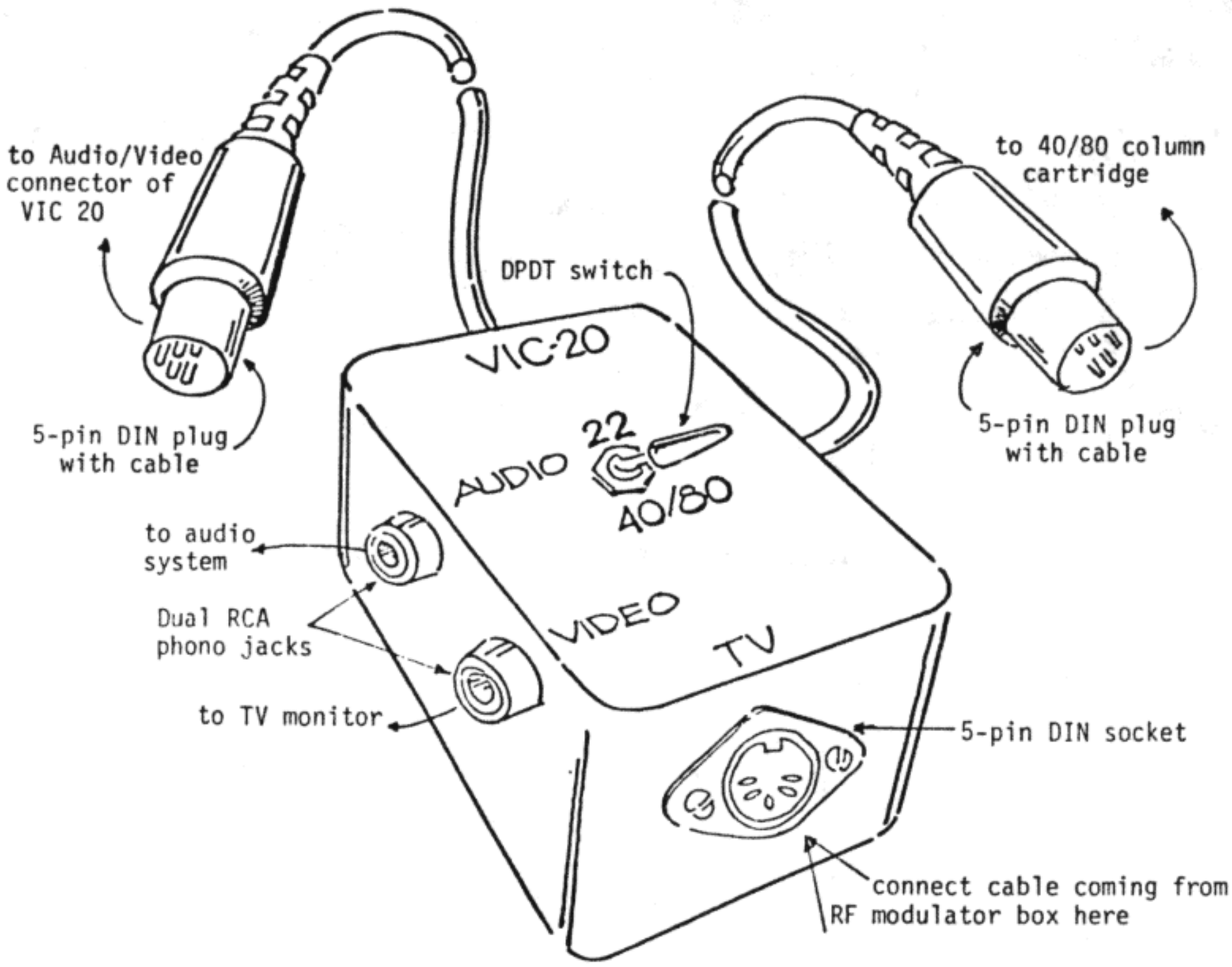


Figure 1

The VIC 20 was originally designed to be connected to a block and white or color television set displaying 22 characters per line. This can be a major limitation to serious users who intend to use their VICs to do wordprocessing or spreadsheets on a 40 or 80 column display. A solution to this problem is to get a 40/80 column adapter such as the one supplied by Data 20. The next thing to consider is the television monitor. With a 40 column display, the regular b&w or color TV set will give a satisfactory display, however, if you are going to use the 80 column display you will need a special TV monitor available in either amber, green, or color from a wide variety of suppliers.

The TV monitor will differ from the regular TV set in the way it is connected to the computer. The signal going to the TV monitor does not need to pass through the RF modulator box (black box with channel 2 or 3 selector switch). You will need a special cable to connect the TV monitor to your VIC 20 or C64. You can construct your own inexpensive cable assembly with the option of retaining the connection to your regular TV set. Figure 1 shows the pin numbering of the Audio/Video connector at the back of the VIC 20 and table 1 shows the purpose of each pin. This connector requires a standard male 5-pin DIN connector (Radio Shack #42-2151 DIN patch cord). The standard connector used by most TV monitors is an RCA jack which will require an RCA plug cable (RS#42-309 audio cable set). Following the circuit diagram in Figure 2 you will have an audio/video adapter which will have 3 connections:

1. 5-pin male DIN plug with cable to be connected to the back of the VIC 20 (RS#42-2151)
2. 5-pin female DIN jack (RS#274-005 5-pin chassis socket) to be connected to the RF modulator cable for regular TV connection
3. 2 RCA jacks (RS#274-332 dual phono jack), one for video output to be connected to the video-in jack of the TV monitor and another for audio output to be connected to the audio-in of a standard stereo set. Both jacks will be connected to the TV monitor and/or stereo set using an RCA plug audio cable set (RS#42-2309)

The two capacitors (C1 and C2) in figure 2 are used to protect the devices connected to the computer. C1 is a coupling capacitor which allows only video signals to pass and prevents any DC (direct current) voltages from passing which can do damage to the TV monitor. C2 is a bypass capacitor which eliminates any high frequency signals, such as noise generated by hair dryers or vacuum cleaners, from interfering with the audio system.

With this adapter and a 40/80 column adapter you can use your VIC 20 to display 22, 40, or 80 columns on either your regular TV set or special TV monitor. The next thing you may get tired of doing is switching from 22 to 40/80 columns which requires that the 5-pin DIN plug coming from the box you just assembled be connected either at the back of the VIC 20 for 22 columns or at the back of the 40/80 column cartridge for 40 or 80 columns. This can easily be simplified by adding a DPDT (double pole, double throw)

switch and another 5-pin DIN plug to the cable adapter circuit. Figure 3 shows how the DPDT switch and the additional cable should be connected. The entire assembly can be housed in a small aluminum box (3 1/4x 2 1/8x 1 1/8) or using RS#270-230 experimentor box.

A simple procedure must be followed to be able to switch from 22 to 40/80 columns display or vice-versa without losing whatever is already in memory. You may find this need for switching when editing BASIC programs. You may find it more convenient to edit using 22 column mode since the maximum line length under this mode is 88 characters while it is only 40 or 80 characters under 40 or 80 column mode.

I will assume you are using a 3 (or more) slot expander together with the 40/80 column cartridge positioned in slot 1 corresponding to DIP switch #1. With the proper cables connected and the mode switch of your audio/video adapter box positioned to 22 columns and DIP switch #1 of the expander board on OFF position (disabling the 40/80 column cartridge), you will see on your monitor a 22 column display (large characters). Load any program you want to test just to see that you wonf lose this program when you switch to 40/80 column mode. To switch to 40/80 column mode follow this procedure carefully in the right sequence:

Step 1. Switch DIP #1 of expander board to ON (establishes the connection of the 40/80 column cartridge to the expansion port).
Step 2. Type SYS40969 (for DATA 20 cartridge), or whatever your 40/80 column cartridge manual tells you to enable 40 column display or SYS40972 (DATA 20 cartridge) for 80 column display. The cursor will disappear at this point. Don't panic.
Step 3. Switch the display mode switch of your audio/video adapter to 40/80 column position. At this point the 'READY' message will appear on the screen in either 40 or 80 column display depending on the choice you made.
Step 4. Now list the program you loaded and it will still be in memory ready to run.

To transfer back to 22 columns all you have to do is switch the DIP #1 to OFF position (disabling the 40/80 column cartridge) then turn the mode switch of your adapter to 22 columns. A big 'READY' message will be displayed at the top of your screen indicating that you are now in 22 column mode. Again, whatever is in memory will be retained. There may be times that you will lose what is in memory, but this can be caused by too much jarring when switching the DIP switch of the expander board. So be extra careful when pushing the DIP switches.

Commodore 64 Users: This project can easily be adapted to the Commodore 64 by just connecting the proper pin numbers of the audio/video connector of the 64. Now you have a choice - 22, 40, or 80 - in just a flip of 2 switches. Happy switching!

Audio Video

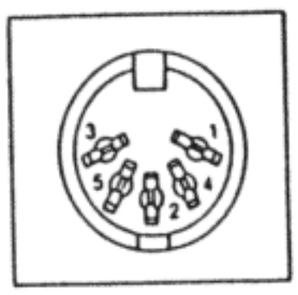


Figure 1.

PIN #	TYPE	NOTE
1	+5V REG	10mA MAX
2	GND	
3	AUDIO	
4	VIDEO LOW	
5	VIDEO HIGH	

Table 1.

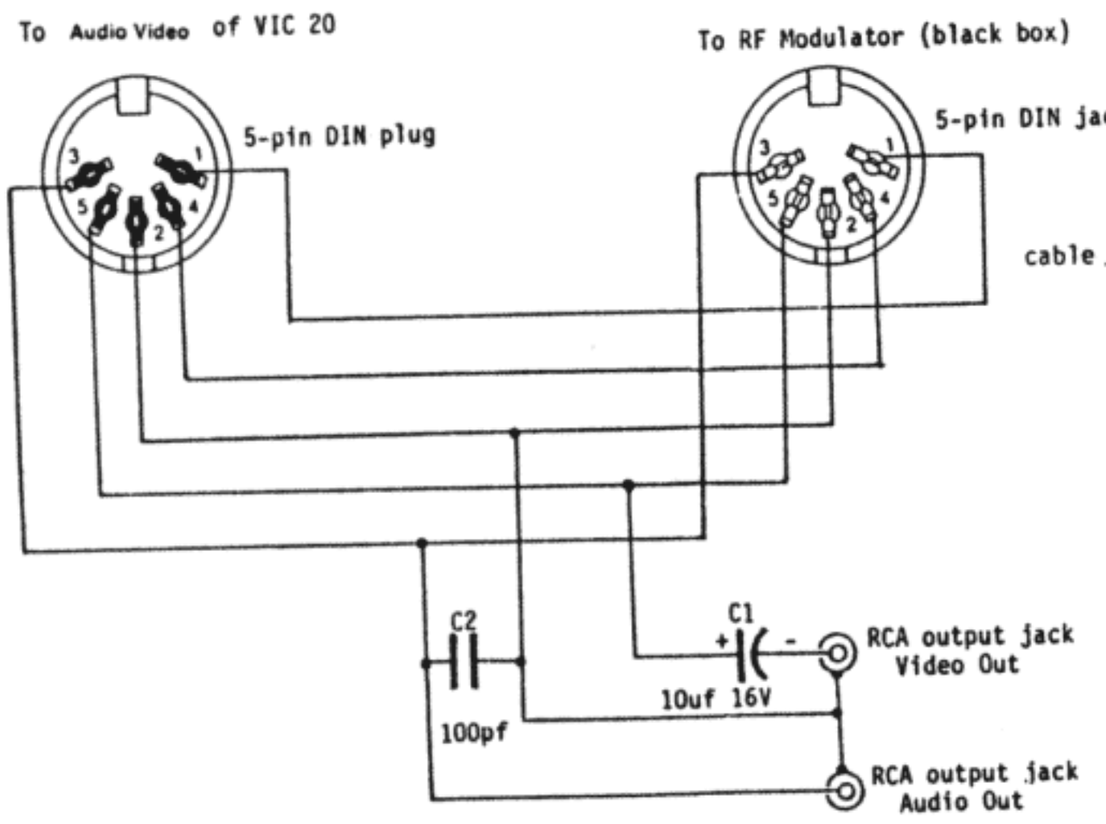


Figure 2: Audio/Video Cable Adapter

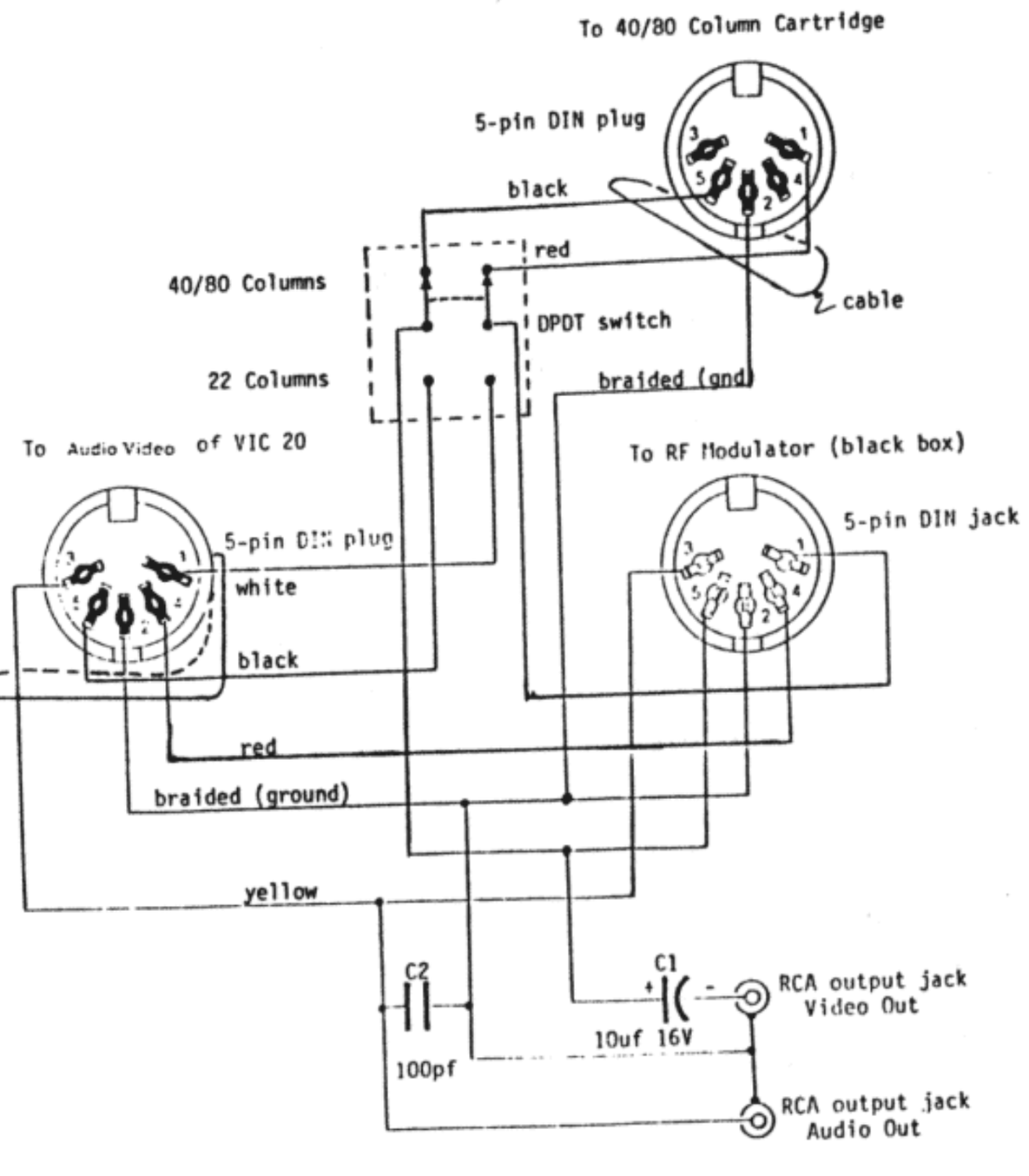


Figure 3: Audio/Video Adapter for 22 or 40/80 Column Display

LINKED LISTS
Part 1

K. Murray Smith
London, Ont.

Sort without Sorting. . .

Linked lists provide a powerful alternative to sorting when large quantities of data must be organized. The following example illustrates some of the features of linked lists. Additional features and some simple possibilities for their use will be found in the summary of Part 2.

The Rentawreck car rental agency is computerizing its operation. In the initial setup it was decided to group the vehicles owned by the company into three lists: those currently rented, those available for rent and those being serviced. Following this, it would be necessary to be able to move vehicles between these lists. For example, when a rented vehicle is returned, it is removed from the rented list and added to the available list (or maybe to the list of those being serviced). When new vehicles are purchased, they must be made available for rental. Also, vehicles that have passed their useful lifetimes should be removed from circulation as they are returned.

It was also a requirement that vehicles on the available list be ranked as to mileage, the vehicle with lowest mileage being the first available, and those on the rented list be ranked by date–to–be–returned.

The Data

The vehicle information is given in a datafile. To refresh your memory, a datafile consists of one or more records. In our case a record contains all the information about one vehicle. The record is made up of one or more fields. Referring to the vehicle record, each datum such as licence plate number is a field.

The datafile is large – the number of vehicles exceeds 500 (although it will be much smaller for our purposes now) and for each is an entry (record) containing plate number, mileage, type of vehicle, engine size, colour, type of transmission, date of last servicing and so on to an estimated 12 fields.

Statement of the Problem

This is easy: write the series of programs necessary to take the company through the automation (so to speak!) of its operation.

Solution to the Problem

This is not as easy. The first step, as any starting problem–solver knows, is to divide the problem into smaller tasks and handle these separately. For us, the tasks become establishing the initial lists of vehicles and providing the means of updating these lists.

Since the lists must exist before they can be manipulated, let us concentrate on this (but keep in mind the requirements of the second task).

Data Storage

Since the results of our initialization program will be required in the second half of the solution, these will be stored on disk (or tape).

Within the program data storage will be in arrays. One might think of using two–dimensional arrays, each row being for a vehicle and each column for one of the dozen fields. However, this is rapidly discarded as we note that it is possible to require the vehicles in a list to be sorted on more than one field (for example, the available list is sorted on mileage but also on vehicle type for the customer who needs a station wagon regardless of its mileage). For each field sorted, an entire array must exist! Unless your minimum memory size is a few hundred kilobytes, this could cause problems.

It is desirable to try to confine our storage to one set of the data. But how can we possibly sort on two or more fields without scrambling the data in the array? The answer is simple – by forgetting traditional sorting techniques.

One of the most common of these involves comparing consecutive elements of an array and switching them if the order is not correct. For example, consider the alphabetic sorting of the array shown in the top line of Figure 1.

Figure 1

	Element Number							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Array Entries	C	F	H	A	E	G	D	B
	C	F	A	E	G	D	B	**H**
	C	A	E	F	D	B	**G**	**H**
	A	C	E	D	B	**F**	**G**	**H**
	A	C	D	B	**E**	**F**	**G**	**H**
	A	C	B	**D**	**E**	**F**	**G**	**H**
	A	B	**C**	**D**	**E**	**F**	**G**	**H**
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**

Comparing elements 1 and 2, we find they are in order.
Comparing elements 2 and 3, we find they are in order.
Comparing elements 3 and 4, we find they must be switched.

This continues until elements 7 and 8 are compared. After this the array looks like the second line of Figure 1. Note that the "largest" value, H, has been placed in its correct spot.

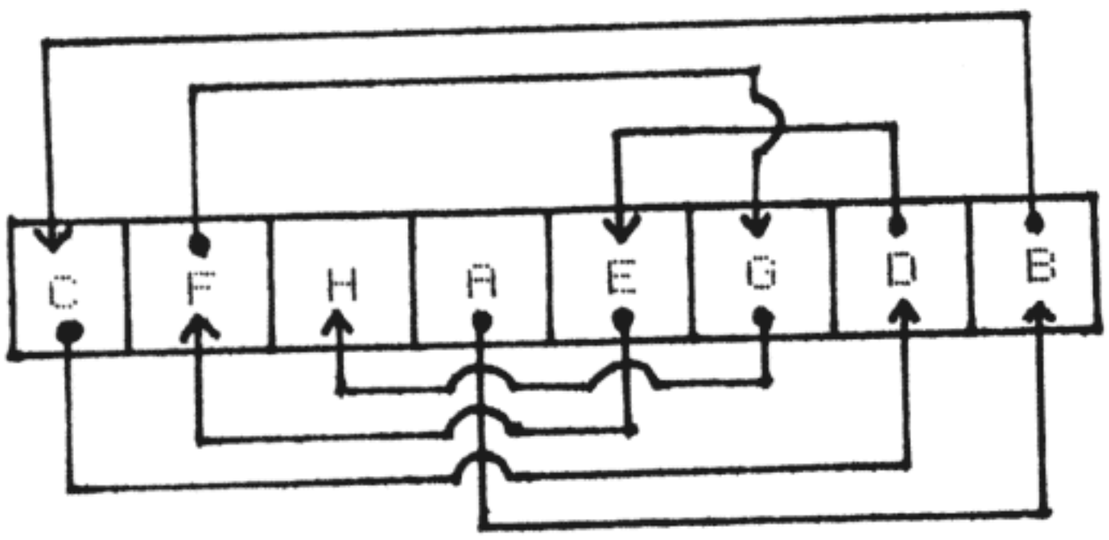
Starting again at elements 1 and 2, the comparisons and switchings are made once more but only up to elements 6 and 7 this time. Now the array looks like the third row. The remaining rows of Figure 1 show the rest of the sorting passes through the array. The underlined elements indicate those that are properly sorted and do not have to be considered on the next pass. Note also that the last two lines are the same. This situation might occur at any time in a sorting and so a check is usually made to see if any switches have been made in a given pass. If not, then the sort is finished.

In our problem this method of sorting would involve the movement of large numbers of elements within the array. Of course you could save many operations by using a one–dimensional array, each element of which is a string containing all the information on one vehicle. However, for our comparisons we are now required to use the MID$ function to compare the correct fields! The number of operations in total then is not significantly reduced, if at all.

Sorting Without Sorting

The problems of sorting on many fields can be solved using linked lists. An example of a linked list is shown in Figure 2. The linkages between array elements are indicated by arrows. The arrows point to the next element in the list, and by following the arrows into and out of each element it is possible to go along the list from beginning to end. (Of course in a large list it would be nice to have a pointer to the start of the list!)

Figure 2



Rather than arrows, a second one–dimensional array of the same length can be used to hold the pointers to the next elements as shown in Figure 3. Note that the quantity STARTER is used to indicate where in LETTERS the list begins. In this case it is the fourth element. Then looking at the fourth element of the LINK array we find an eight. This says the next letter in the list is the eighth element of LETTERS (which is B). Then going for the eighth element of LINK, it is a one, giving C as the next letter in the list and so on. Since H is the last letter in the list, its corresponding element of LINK must indicate somehow that the list is done and a zero is used (arbitrarily) for this purpose.

Figure 3

ARRAY	Element Numbers							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
LETTERS	C	F	H	A	E	G	D	B
LINK	7	6	0	8	2	3	5	1
STARTER	4							

Regardless of the number of fields in each record, only one LINK array is needed. This array is valid only for the specific field we linked. If we wish a file organized on more than one field, we will need one array of pointers for each field to be linked. (A file containing four fields per record and sorted on all fields would require four arrays for the file and then four for the pointers whereas traditional sorts might require up to four arrays for the file plus 16 more, one group of four for each field sorted.)

The Mechanics

Let's now look at the mechanics of setting up the linked list. Table 1 shows some of Rentawreck's currently rented vehicles (only two fields are shown).

Table 1: The Rented Vehicles

Plate Number	Due Date (Month–Day)
TER 686	12–07
LAS 241	12–18
JDL 413	12–11
VAX 750	12–27
SJS 017	12–03
SPY 007	12–11

Assume that there is a possibility of a maximum of 50 vehicles. This provides a limit on our array sizes right now. Program 1 reads the data and links the file based on due date and Table 2 describes the variables used.

Program Description

The following is a line–by–line description of Program 1.

30,40: dimension the arrays to be used. Only line 30 need be changed to affect the array sizes throughout the program.

60–90: by filling the arrays with these strings, the arrays can be printed beside each other in columns.

110–160: input the plate numbers and due dates for the vehicles currently rented, keeping a count of how many in NR. Stop reading data when a 'dummy' record is read (line 140).

180–200: copy into a work array the portion of the due date array to be linked. This is necessary as the subroutine which sets up the links destroys the array upon which it works.

210,220: define the highest and lowest elements in the work array to be used in the linking.

230: perform the linking.

240: save the entry point to the linked list of rented vehicles.

8000: the I–loop ensures that we go through the portion of the working array we are using once for each element.

8010,8020: since we are linking values from smallest to largest, here we assume that the smallest value is in the first element of the portion of the array we are using. SM$ and SB store this "smallest value so far" and its subscript.

8030–8080: go through the remainder of this portion of the work array and see if there is a smaller value – if so, put this value in SM$ and the subscript into SB.

8090–8120: check if this is the first pass through this section of the work array – if so, then we have the entry point to this linked list (SE). This subscript must also be saved (in J) because this element of the link array (LD) must contain the pointer to the next smallest value.

8140–8150: on any other pass through the work array, put the subscript of the current smallest value into the correct position of the pointer array (as given by J) and update the value of J.

8160: replace the current smallest value by '[' which has an ASCII value higher than any digit or uppercase letter. This ensures that this element will not be chosen again as a next–smallest value.

8180–8190: if we have now linked all of the values in the work array, set the last pointer to a zero to indicate the end of the list.

Table 2: Description of Variables

DU$	the array containing the due dates of the rented vehicles
I,J,K	loop controllers (see program description)
LD	the link array – contains the pointers
NR	the number of rented vehicles
PL$	the array containing licence plate numbers
S	the size (dimension) of all arrays
SB	the subscript of the smallest element found in the work array so far
SE	where to enter this particular linked list
SM	the starting element number for a list to be linked
SM$	the smallest value found in the work array up to this point
SR	the pointer to the starting element for the linked list of rented vehicles
UL	the highest element number for a list to be linked
WO$	a work array, starting as a copy of the list to be linked

Program Output

Since the job of organizing Rentawreck's files has just begun, the program contains no output section. The arrays can be examined by typing in immediate mode:

```
PRINT SR
A$ = "/"
FORI = 1TOUL:?PL$(I)A$DU$(I)A$LD(I):NEXTI
```

(For output to printer, precede the above instructions with OPEN4,4:CMD4 and follow them with PRINT#4:CLOSE4)

You should see:

```
 5
TER 686  12–07 3
LAS 241  12–18 4
JDL 413  12–11 6
VAX 750  12–27 0
SJS 017  12–03 1
SPY 007  12–11 2
```

The "sorted" listing of due dates begins with element 5, the fifth pointer shows the next due date in element 1, the first pointer shows the next due date in element 3 and so on. The second last due date is element 2 and element 2 of the link array is 4. The fourth element of the link array is 0, indicating the end of the list (those memory scavengers out there who like to use the zero elements of arrays might use –1 to indicate the end!)

The Other Lists

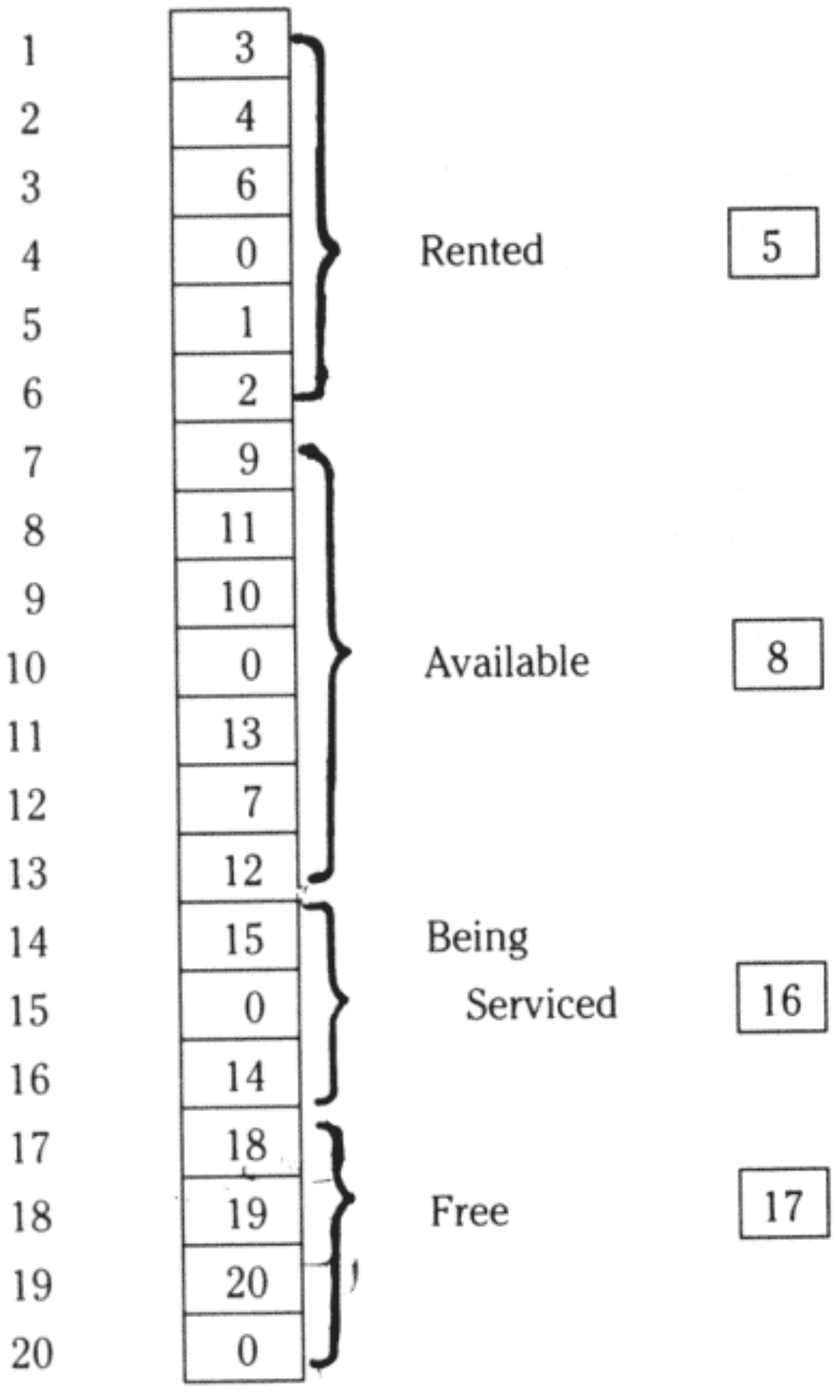
Part of out initial task is done. We have yet to establish linked lists for those vehicles which are available for rental (linked by mileage) and for those being serviced (linked by mileage also). Where should we store these data? We could use another set of arrays for each of these situations. We would then have three separate pointer arrays. The result would be well–organized data, each category having its own linkage array.

At this point it would be wise to remember that we earlier said that although we were working to solve the first half of the problem, we should also keep in mind the requirements of the second part. One of these is that we are able to transfer vehicles between lists as rented ones are returned, serviced ones are made available and so on. With separate lists, elements will have to be copied from one set of arrays to another. This also implies that each set of arrays must be dimensioned sufficiently large to hold nearly all (if not all) of the vehicles at once. Suddenly memory usage is increasing again, a situation we wished to avoid. In addition, what do we do with the newly–vacated array elements? How do we keep track of them to use them again later?

The best place to store the other vehicles is in the same array as those currently rented. The advantage here is that all of the vehicles owned by the company are together in the same place. Not only that but the pointers for all three lists can also be kept in the same array, each list having its own entry value and terminating with a zero element.

Figure 4 shows a LINK array containing pointers for the three linked lists previously described. Also linked into a list are the unused (or free) elements of the array. These are useful for storing data of eventual purchases. It is important to realize that the lists of rented and available vehicles were sorted on different fields, namely due date for the rented and mileage for the available. This is a very powerful feature of linked lists.

Figure 4



Program 2 contains additions to Program 1 to read the data for the available vehicles and those being serviced, to link both of these by mileage, and finally to link the free space together. Linking the free space provides for good memory management. As vehicles are sold, the array space occupied by these can be reused by new vehicles purchased.

Lines 1020 and 1060 require an explanation. The mileages are stored as strings for two reasons. One is the desire to have these numbers displayed or printed right–justified in a column, that is, lined up along the right ends of the numbers. The second is being able to use the same routine for linking all of the lists. Since the maximum mileage is 999999, line 1060 is used with line 1020 to put leading blanks in mileages having fewer than six digits. Otherwise '11' would sort as being greater than '100000'!

Table 3: Description of Additional Variables

B$	a string of six blanks to provide leading blanks where necessary in the mileage strings
MI$	the array containing the mileages (in kilometres) of the vehicles
NA,NS	the numbers of available vehicles and vehicles being serviced, respectively
SA,SS,SF	the pointers to the starting elements of the linked lists of the available vehicles, those being serviced and the free elements respectively.

To see the first three linked lists and the start of the linked free list, type in immediate mode again:

```
PRINT SR;SA;SS;SF
A$ = " / "
FORI = 1TO20:?PL$(I)A$DU$(I)A$MI$(I)A$LD(I):NEXTI
```

You can then start at the element given by SD, SA, SS or SF to enter the appropriate list and then follow it through to the end (as indicated by a zero pointer). If you wish to see how the free list ends, then replace the

'1 TO 20' by '21 TO 50'

(The output is best seen on a printer using FORI = 1TO50 with the opening and closing statements given previously.)

Saving the Lists

In order to be able to update the lists in Part 2 of this article, we need to save these lists on tape or disk. Program 3 contains the necessary additions to the current program to save the data in a sequential file on disk.

Summary

Thus far we have accomplished all of the requirements for the initial setup of Rentawreck's files. We have also attempted to be conservative with respect to memory and have given some consideration to being able to easily manipulate the lists, something that we must be able to do in Part 2.

• Program 1 and Program 2 on NEXT page •

Program 3

```
10 rem– program 3
4000 rem– output linked lists to disk
4010 print " do you wish to save the linked "
4020 print " lists at this time? "
4030 print " (type y or n): " ;
4040 get d$
4050 if d$ = " " then 4040
4060 print d$
4070 if d$ = " n " then 7999
4080 if d$<> " y " then 4030
4090 c$ = chr$(13)
4100 open1,8,2, " 0:linked lists,seq,w "
4110 print#1,s;c$;ul;c$;sr;c$;sa;c$;ss;c$;sf
4120 for i = 1 to s
4130 :print#1,pl$(i);c$;du$(i);c$;mi$(i);c$;ld(i)
4140 next i
4150 close1
4160 print tab(10); " lists have been saved "
```

Program 1

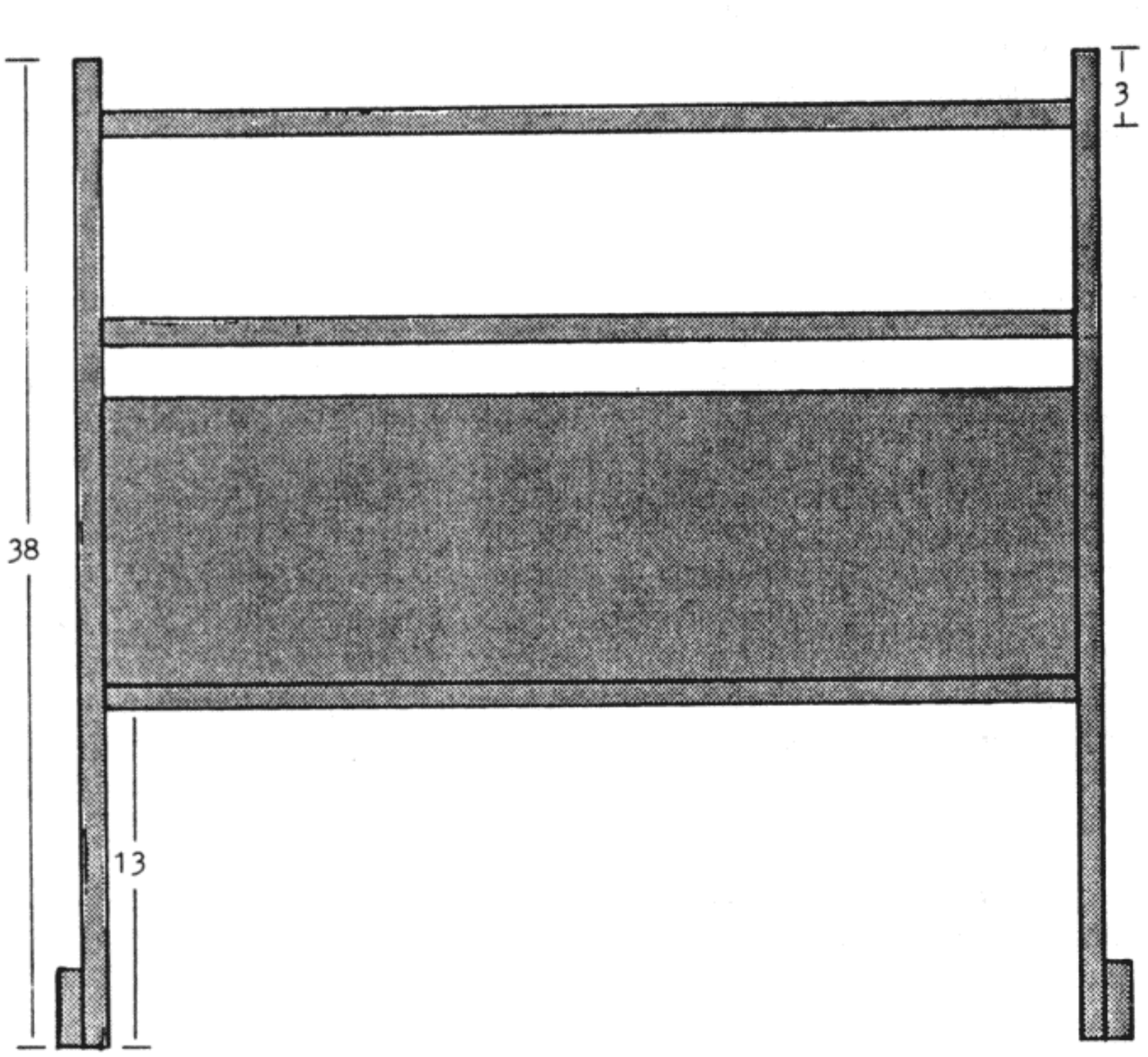
```
10 rem– program 1
20 rem– set up arrays, variables
30 s = 50
40 dim pl$(s),du$(s),wo$(s),ld(s)
50 rem– initialize arrays
60 for i = 1 to s
70 :pl$(i) = "              " :rem 7 spaces
80 :du$(i) = "          " :rem 5 spaces
90 next i
100 rem– input arrays
110 nr = 0
120 for i = 1 to s
130 :read pl$(i),du$(i)
140 :if pl$(i) = "dum num" then 170
150 :nr = nr + 1
160 next i
170 rem– read due dates into work array
180 for i = 1 to nr
190 :wo$(i) = du$(i)
200 next i
210 ul = nr
220 sm = 1
230 gosub 8000
240 sr = se
7999 end
8000 i = sm
8010 :sm$ = wo$(sm)
8020 :sb = sm
8030 :k = sm + 1
8040 :if sm$< = wo$(k) then 8070
8050 ::sm$ = wo$(k)
8060 ::sb = k
8070 ::k = k + 1
8080 :if k< = ul then 8040
8090 :if i<>sm then 8140
8100 :rem– establish entry point to the link array
8110 :se = sb
8120 :j = se
8130 :goto 8160
8140 :ld(j) = sb
8150 :j = sb
8160 :wo$(sb) = " [ "
8170 :i = i + 1
8180 if i< = ul then 8010
8190 ld(j) = 0
8200 return
10000 data ter 686,12–07,las 241,12–18
10001 data jdl 413,12–11,vax 750,12–27
10002 data sjs 017,12–03,spy 007,12–11
10003 data dum num,99–99
```

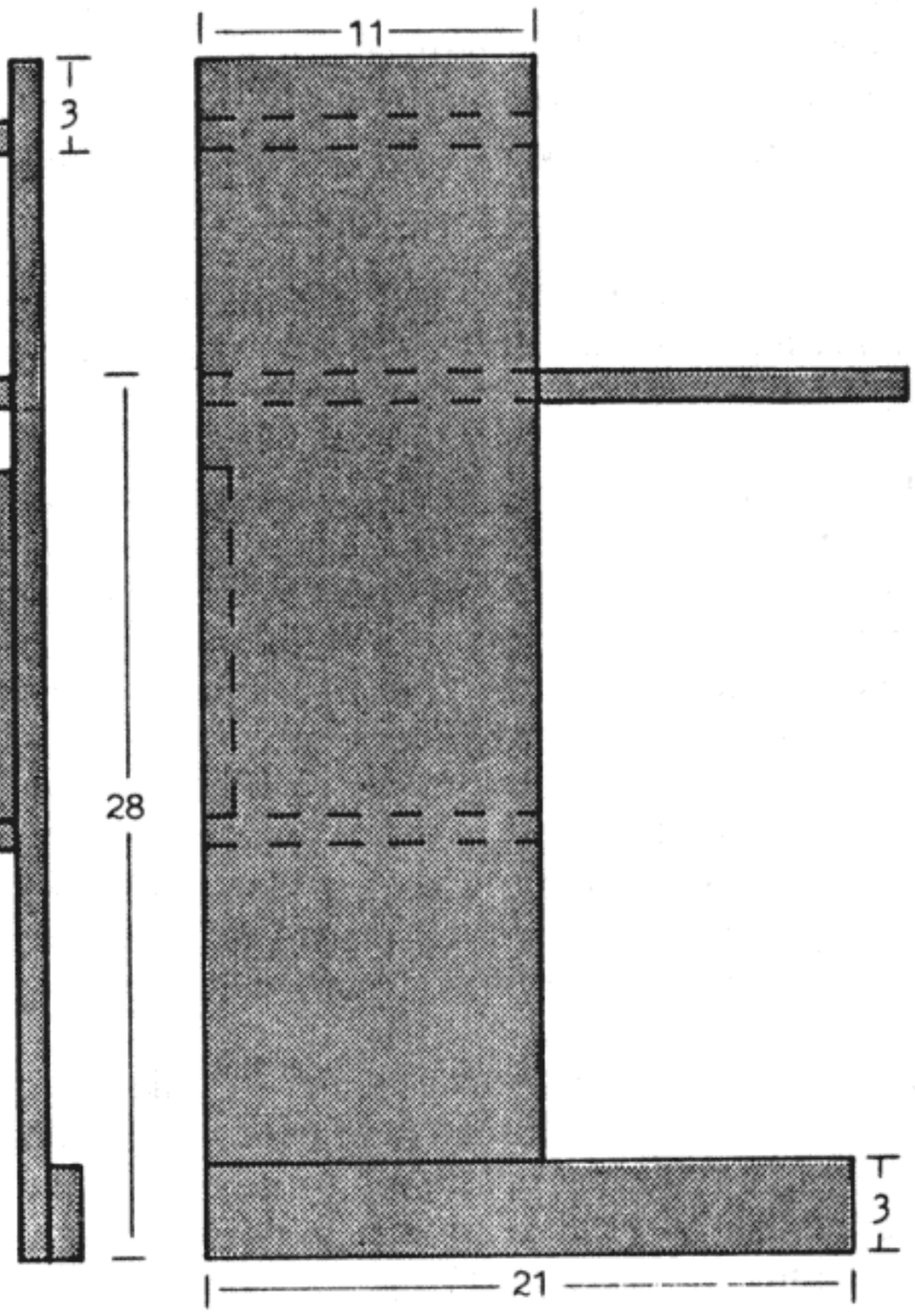
Program 2

```
10 rem– program 2
40 dim pl$(s),du$(s),wo$(s),mi$(s),ld(s)
85 :mi$(i) = "              " :rem 6 spaces
140 :if pl$(i) = "dum num" then 165
150 :nr = nr + 1
165 rem– blank out the dummy data
166 pl$(i) = "              " :rem 7 spaces
167 du$(i) = "          " :rem 5 spaces
1000 rem– for available vehicles
1010 na = 0
1020 b$ = "              " :rem 6 spaces
1030 for i = nr + 1 to s
1040 :read pl$(i),mi$(i)
1050 :if pl$(i) = "dum num" then 1090
1060 :mi$(i) = left$(b$,6–len(mi$(i))) + mi$(i)
1070 :na = na + 1
1080 next i
1090 rem– blank out the dummy data
1100 pl$(i) = "              " :rem 7 spaces
1110 mi$(i) = "              " :rem 6 spaces
1120 rem– set up lower and upper limits on this part of array
1130 sm = nr + 1
1140 ul = nr + na
1150 rem– read mileage into working array
1160 for i = sm to ul
1170 :wo$(i) = mi$(i)
1180 next i
1190 gosub 8000
1200 sa = se
2000 rem– for vehicles being serviced
2010 ns = 0
2020 b$ = "              " :rem 6 spaces
2030 for i = nr + na + 1 to s
2040 :read pl$(i),mi$(i)
2050 :if pl$(i) = "dum num" then 2090
2060 :mi$(i) = left$(b$,6–len(mi$(i))) + mi$(i)
2070 :ns = ns + 1
2080 next i
2090 rem– blank out the dummy data
2100 pl$(i) = "              " :rem 7 spaces
2110 mi$(i) = "              " :rem 6 spaces
2120 rem– set lower and upper limits
2130 sm = nr + na + 1
2140 ul = nr + na + ns
2150 rem– read mileage into work array
2160 for i = sm to ul
2170 :wo$(i) = mi$(i)
2180 next i
2190 gosub 8000
2200 ss = se
3000 rem– link free space
3010 i = ul + 1
3020 sf = i
3030 if i = s then 3070
3040 ld(i) = i + 1
3050 :i = i + 1
3060 if i< = 49 then 3040
3070 ld(s) = 0
10004 data los 190,101760,njz 242,1569
10005 data kfd 822,137564,tnh 487,200147
10006 data zom 434,6572,eam 876,87624
10007 data upx 814,16583,dum num,–1
10008 data ere 098,60105,trk 633,120354
10009 data gbd 484,32900,dum num,–1
```

Computing Desk

Scott Johnson
Traverse City, MI





For many of us, the purchase of our first computer was the culmination of months, or perhaps years, of longing and dreaming. We hurried home from the store with our dream tucked under our arm, safely packed away in cardboard and styrofoam. With great anticipation we finally removed our dream realized from its plastic cocoon, only to be faced with another major decision; where are we going to put it? On the kitchen table? The living room floor? On that rickety old card table? I'm afraid that too many home computers have either ended up gathering dust tucked away behind the family TV, or stashed in the top of a closet along with other dreams that we didn't have a convenient place for.

Well, rescue that Commodore from the dark corner it's hiding in because here's a handy little computer desk that almost anyone can build from less than one sheet of 3/4 plywood. This simple desk will easily hold your computer, monitor, disk drive, cassette deck, printer, and maybe even have some room left for your back issues of The Transactor.

The desk is constructed from just 8 pieces of plywood;

(5) 3/4 x 11 x 38
(1) 3/4 x 23 x 38
(2) 3/4 x 3 x 21

All of the joints may be just carefully glued and nailed together, or if you have access to the proper equipment, you might try dadoing the sides before attaching the three horizontal shelves.

Good luck, we hope that your computer will bring you many more hours of computing enjoyment now that you, and it, have a simple and convenient desk at which to work, play, and dream.

Rethinking DATAfication

Nick Sullivan
Scarborough, Ont.

Every now and then we are faced with the necessity of converting memory into DATA statements, whether it's sprite descriptions, machine language, character sets, or something else. Most of us have written or otherwise obtained programs to do this fiddly job for us.

Even so, it's hard to be thrilled about the result. DATA statements take up a lot of memory relative to the data they actually embody, and the process of poking the data back into memory is a slow one. The compensating advantage, of course, is that data keeps company with a BASIC program more comfortably in this form than in any other.

It would be nice, wouldn't it, to preserve that advantage while minimizing the drawbacks? It was with that thought that I wrote the DATAfier.

The DATAfier also creates DATA statements, and links them into memory in the usual way with the dynamic keyboard technique. But instead of being encoded as decimal numbers separated by commas, in this case each byte of data is represented by two letters, and the pairs of letters are concatenated without commas into strings up to 74 characters in length.

Consider how that saves memory. One byte of data encoded decimally requires, on average, more than two and a half bytes of memory, plus another byte for the comma. Encoding that same byte as a pair of letters results in a saving of nearly 50 per cent.

Along with the DATA statements, the DATAfier also provides a short machine language routine to poke the encoded data into memory. To get an idea of its speed I once encoded the entire C64 BASIC ROM, all 8192 bytes of it, a process that took several minutes to complete. The DATAfier poked it all back in less than four seconds. Try that with decimally encoded DATA some time and see how you do.

Using The DATAfier

Type RUN, and RETURN.

The program will prompt you with: "D/DISK M/MEMORY? " Reply with D if the program you are intending to datafy is on disk (drive 0 only). You will be asked to enter the filename, then the Datafier will do the rest.

If the program is in memory, type M. The Datafier will prompt you with:

"START, END? "

Enter the start and end addresses in hexadecimal, just as you would with the Machine Language Monitor, but separate the numbers with a comma. Remember that the end address you give should be one byte beyond the area you want DATAfied. Press RETURN.

Now the DATAfier goes to work, and won't stop till it's done. You'll see things happening on the screen — that's the dynamic keyboard technique in action.

When finished, the first 37 lines (0–36) are erased by the subroutine at line 31. Deleting lines can be accomplished using a variation of the same technique used to enter lines. However, the technique used in this subroutine eliminates the trouble of deleting each line individually. First it tracks the line link pointers for 37 lines. Then it adjusts the link pointer of the first line so that it points to the link pointer of line 8997. By simply erasing line 0, BASIC will remove all the bytes up to the end of line 36. Just before this is done, line 36 adjusts the high byte of the End-of BASIC pointer as the editor does not account for a move across more than one page boundary.

The result will be a subroutine beginning at line number 9000. You can merge the subroutine into any program that requires the data (one way to do the merge is with the TransBASIC ADD command — see elsewhere in this issue). The instruction GOSUB 9000 will do the work of reading the DATA statements and poking them into memory. Naturally, if you have any other DATA statements preceding line 9000 in your program they must already have been read at the time the GOSUB 9000 instruction is given.

The method of encoding the data as a pair of alphabetic characters is extremely simple: each character represents one nybble of the hexadecimal data byte. The letter 'A' represents 0; each subsequent letter represents a subsequent hexadecimal digit. 'AA', therefore, encodes a zero byte; 'DM' would be equivalent to 3*16+12 — $3C, or 60. The highest letter used is P, which represents the hexadecimal digit F.

The only disadvantage of DATAfied DATA is that it is even harder to read than its decimal counterpart. However, the DATAfier wasn't designed for publishing programs for hand entry. It's main purpose is for including your machine language subroutines as part of your BASIC text. Using any type of loader will make your hybrid programs more transportable, but DATAfied DATA is a viable alternative for programs that will almost never be 'transported' by hand.

Editor's Note

Between the time the TransBASIC article was prepared and this one, Nick has become the new Editor for TPUG Magazine.

One last note about the program. . . you'll notice several spaces in the middle of line 15. The reason? Line 15 prints variable definitions at the bottom of each screen during the DATAfication process. This will always take two lines. When BASIC prints a variable, the first character output is a 'Cursor Right'. If that cursor right character is the one responsible for moving onto the second line, the screen line wrap table will not be updated so that the two lines are treated as one line that has 'wrapped around'. When the Carriage Return is dropped onto this string, the GOTO command will not be executed unless a second Carriage Return is sent. To avoid this, it is necessary to have the second line entered as the result of Spaces. Depending on the column width of your particular machine, these spaces will need to be inserted to ensure the two lines become one. An article on the screen wrap table will be appearing in a not too distant Transactor. M.Ed.

Converting The Datafier For Other Commodore Machines

As shown, the Datafier will work on the Commodore 64 only. To use it with other machines changes must be made.

Line 7 contains the instruction G=35, which sets the number of memory bytes to be encoded on each DATA line. This depends on the screen size, and for other machines should be set as follows:

VIC–20	G=39
40–Column PETs	G=34
80–Column PETs	G=34

For the VIC–20 make the following changes in line 7:

K=D+8	becomes K=D+3
POKE 198,10	becomes POKE 198,5

Again for the VIC–20, change P+9 in line 8 to P+4.

Other changes in the first part of the program (before line 9000) affect BASIC 2.0 and BASIC 4.0 machines only, not the VIC–20. The changes are:

Lines 7, 23, 24:	Change 198 to 158
Lines 8, 23, 24:	Change 631 to 623
Line 15:	Change 152 to 174
Line 24:	Change 632 to 624
Line 30:	Change W=43 to W=40

The remaining changes affect ROM references in the machine language encoded as DATA statements beginning at line 9012. The replacement DATA lines are as follows:

VIC–20:

```
9007 data  32, 253, 206,  32, 158, 205
9008 data  32, 163, 214, 133,  34, 162
9017 data 104, 104,  96,  76,   8, 207
```

BASIC 2.0:

```
9007 data  32, 248, 205,  32, 159, 204
9008 data  32, 125, 213, 133,  34, 162
9014 data 240,  16, 177,  31, 201,  81
9017 data 104, 104,  96,  76,   3, 206
```

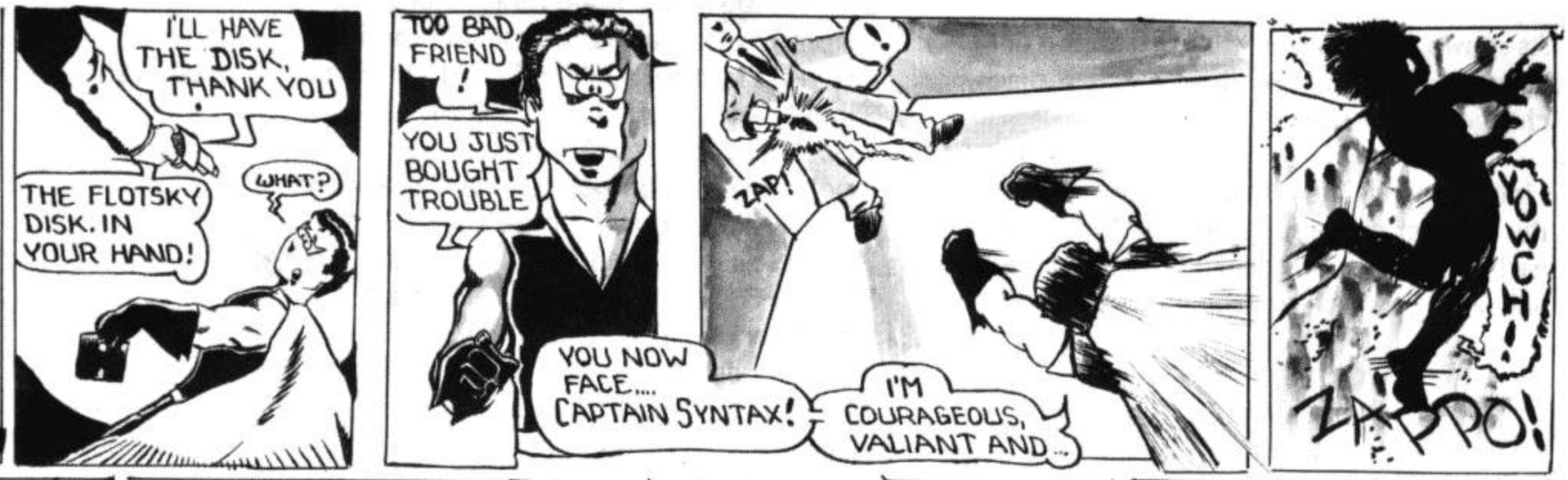
BASIC 4.0:

```
9007 data  32, 245, 190,  32, 152, 189
9008 data  32, 181, 199, 133,  34, 162
9014 data 240,  16, 177,  31, 201,  81
9017 data 104, 104,  96,  76,   0, 191
```

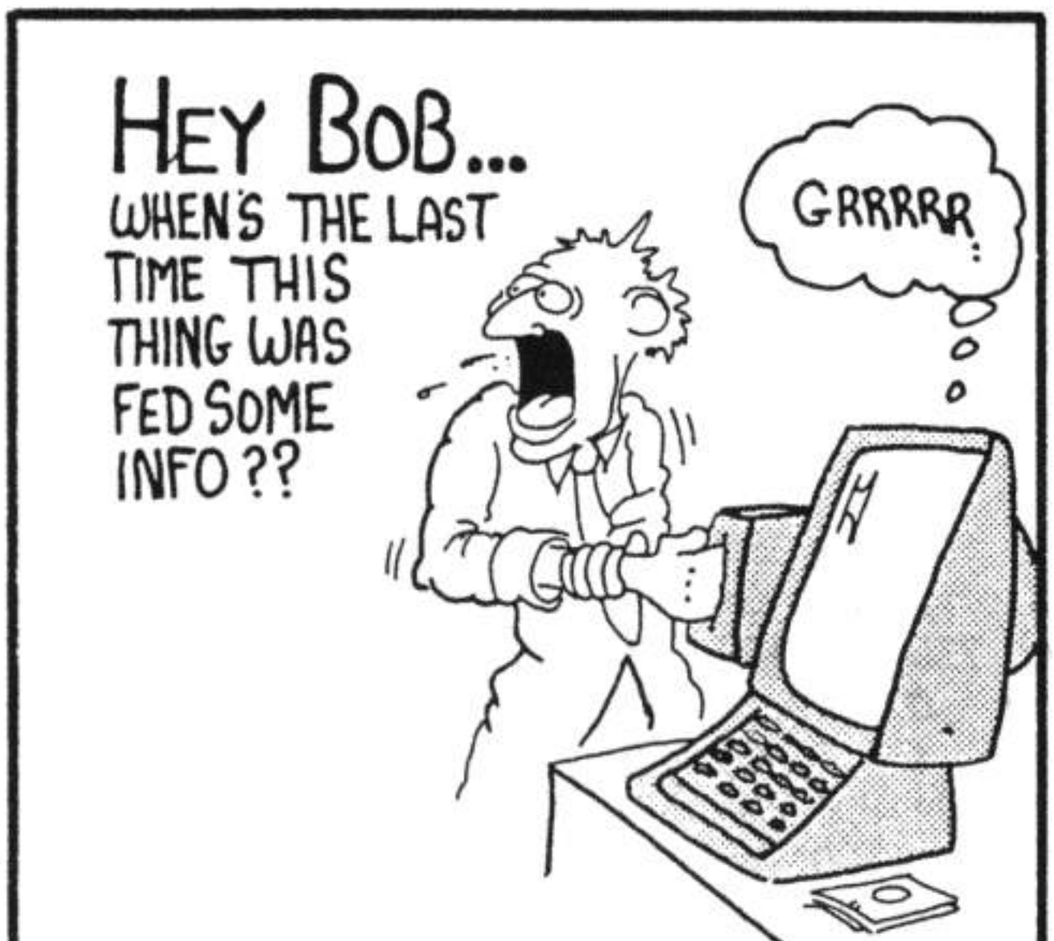
The DATAfier

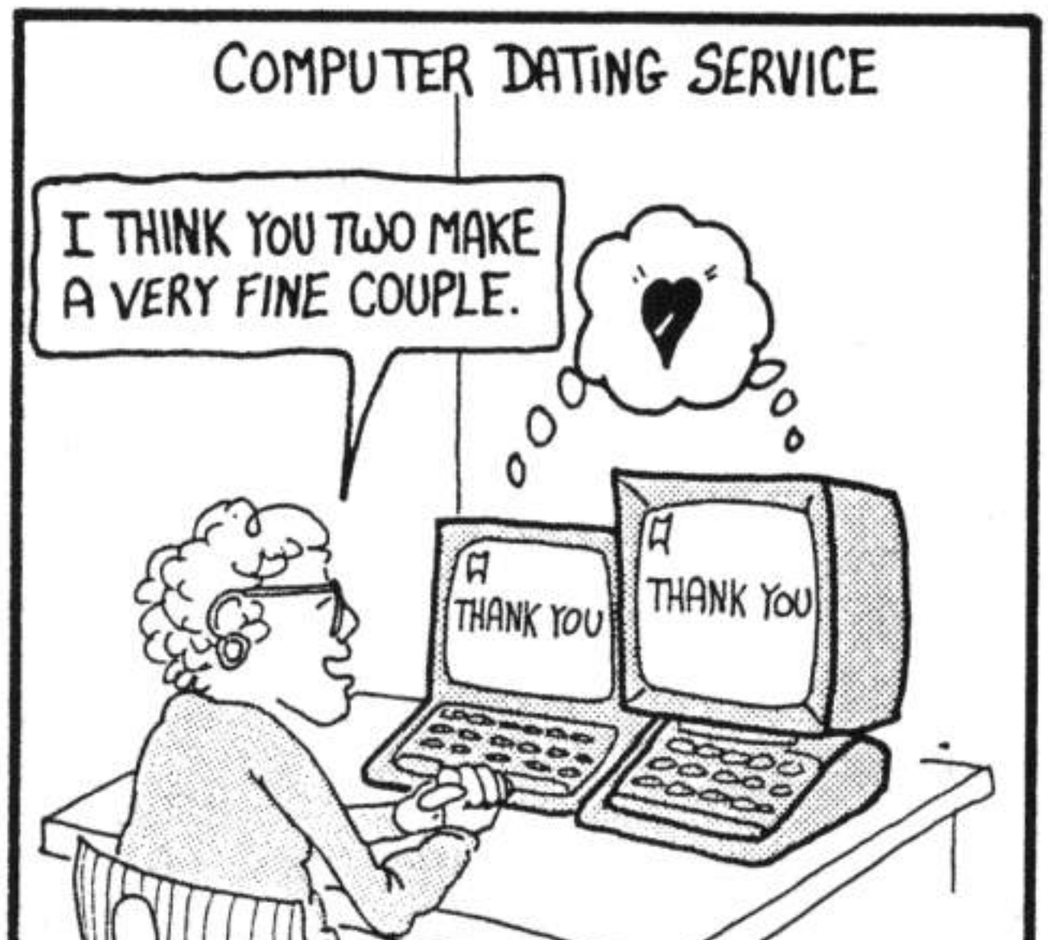
```
0 input "■d/disk  m/memory";dm$
1 if dm$<>"d" and dm$<>"m" then end
2 d=9020 : dm=1 : if dm$="d" then dm=2 : gosub 25 : goto 5
3 input "qstart, end (hex)";a$,b$
4 gosub 16 : a=u : a$=b$ : gosub 16 : b=u
5 poke 252,a/256 : poke 251,a-256*peek(252)
6 if b<=a then print "range error" : end
7 g=35 : k=d+8 : poke 198,10 : print "■Sqq■"
8 for p=631 to p+9 : poke p,13 : next : c=a : for i=d to k : if c=b or s goto 13
9 print i;chr$(157);"data"; : for j=1 to g
10 m=peek(c) : if dm=2 then get#2,m$ : m=asc(m$+chr$(0)) : s=st
11 z=m-16*(int(m/16)) : print chr$(65+m/16)+chr$(65+z);
12 c=c+1 : y=-i*(c=b or s>0) : if c<b and s=0 then next
13 print : next i : if y then i=y
14 t=7 : if c=b or s then t=19
15 print"a="c":b="b":d="i"[add spaces]:s="s":dm="dm"
   :poke152,2:goto"t"■s■": end
16 u=0 : a$=right$("0000"+a$,4)
17 for i=1 to 4 : v=asc(mid$(a$,i,1))
18 v=v-48+7*(v>64) : u=u+v*16↑(4-i) : next : return
19 d$=str$(d-9020) : q$=chr$(34) : close 2 : close 1
20 a$="9002 poke 251,"+str$(peek(251)) : b$="9003 poke 252,"
   +str$(peek(252))
21 c$="9004 for i=0 to"+d$+" : read a$" : print "■Sqqq■" a$ : print b$
   : print c$
22 print "u=0 : goto 24■s■"
23 for i=631 to i+3 : poke i,13 : next : poke 198,4 : end
24 print "■Sqqq■0" : poke 631,13 : poke 632,147 : poke 198,2
   : gosub 30 : print "■s■" : end
25 input "■qfilename";fl$ : if fl$="" then end
26 open 1,8,15 : open 2,8,2,fl$+",p,r"
27 input#1,de,de$,df,df : if de>19 goto 29
28 get#2,a$,b$ : z$=chr$(0) : a=asc(a$+z$)+256*asc(b$+z$) : b=65536
   : return
29 print "■qdisk error: " de$ : end
30 w=43 : x=w
31 for i=1 to 37
32 w=peek(w)+256*peek(w+1) : if i=1 then y=w
33 next
34 poke y,peek(w)
35 poke y+1,peek(w+1)
36 poke x+3,peek(x+3)-int((w-y)/256):return
8997 :
8998 end
8999 :
9000 for i=319 to 384
9001 read a : poke i,a : next
9005 sys 319,a$ : next : return
9006 :
9007 data  32, 253, 174,  32, 158, 173
9008 data  32, 163, 182, 133,  36, 162
9009 data   0, 160, 255,  32, 102,   1
9010 data  10,  10,  10,  10, 133,  37
9011 data  32, 102,   1,   5,  37, 129
9012 data 251, 230, 251, 208, 236, 230
9013 data 252, 208, 232, 200, 196,  36
9014 data 240,  16, 177,  34, 201,  81
9015 data 176,  13, 201,  65, 144,   9
9016 data 233,   1,  41,  15,  16,   2
9017 data 104, 104,  96,  76,   8, 175
9018 :
```

CAPTAIN SYNTAX



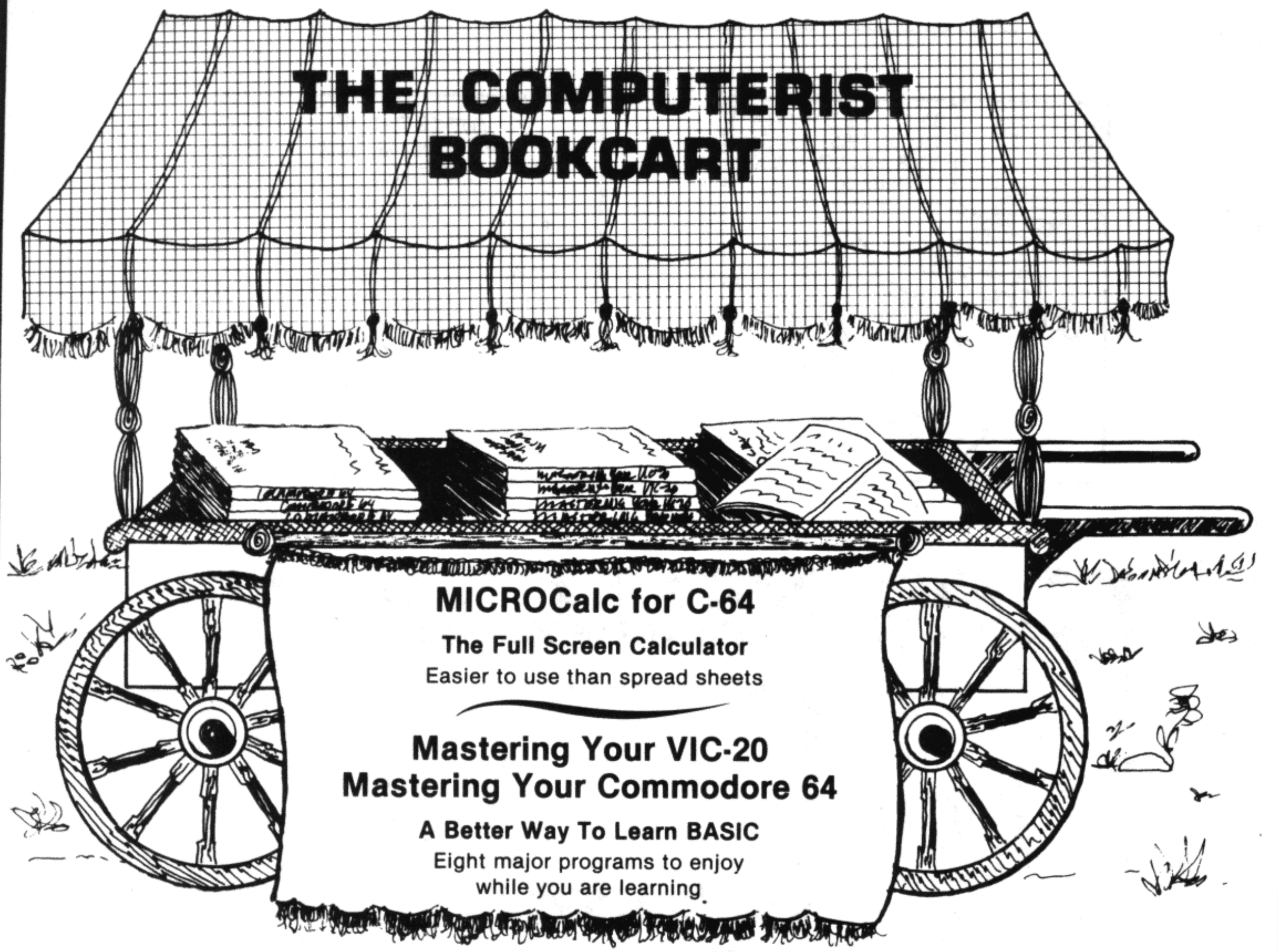
©DAN SLOAN-84





ELMO MELLONHEAD

THE COMPUTERIST BOOKCART



MICROCalc for C-64

The Full Screen Calculator

Easier to use than spread sheets

Mastering Your VIC-20
Mastering Your Commodore 64

A Better Way To Learn BASIC

Eight major programs to enjoy
while you are learning

Mastering Your VIC-20
Mastering Your Commodore 64

The 8 programs, "run-ready" on disk (C-64) or tape (VIC-20) and explained in the 160-192 page book, each demonstrate important concepts of BASIC while providing useful, enjoyable software. Programs include:

- Player — compose songs from your keyboard, save, load and edit for perfect music
- MicroCalc — display calculation program that make even complex operations easy
- Master — a one or two person guessing game
- Clock — character graphics for a digital clock

VIC-20 with tape & book ..... **just $19.95**
C-64 with disk & book (avail. Sept.) ..... **just $19.95**

Look for us at the
International Software Show
Toronto, September 20-23

MICROCalc for C-64

This on-screen calculator comes with diskette and 48-page manual offering a wide variety of useful screens, and a great way to learn BASIC expressions if you don't already know them.

- Unlimited calculation length & complexity
- Screens can be linked and saved on disk/cassette
- Build a library of customized screens
- Provide formatted printer output

Diskette & 48-page manual ..... **just $29.95**

For the Freshest Books, Buy Direct!

- No prehandled books with bent corners
- Books come direct to your door
- No time wasted searching store to store
- 24 hours from order receipt to shipment
- No shipping/handling charges
- No sales tax (except 5% MA res.)
- Check, MO, VISA/MC accepted (prepaid only)

The Computerist Bookcart
P.O. Box 6502, Chelmsford, MA 01824
For faster service, phone: 617/256 – 3649.

COMMODORE OWNERS

Join the world's largest, active Commodore Owners Association.

- Access to thousands of public domain programs on tape and disk for your Commodore 64, VIC 20 and PET/CBM.
- Monthly Club Magazine
- Annual Convention
- Member Bulletin Board
- Local Chapter Meetings

Send $1.00 for Program Information Catalogue.
(Free with membership).

Membership Fees for 12 Months	Canada	—	$20 Can.
	U.S.A.	—	$20 U.S.
	Overseas	—	$30 U.S.

T.P.U.G. Inc.
Department "M"
1912A Avenue Road, Suite 1
Toronto, Ontario, Canada M5M 4A1

* LET US KNOW WHICH MACHINE YOU USE *

100,000 CHOOSE COMAL 50,000 USERS‡

(1) DISK BASED COMAL Version 0.14

- COMAL STARTER KIT—Commodore 64™ System Disk, Tutorial Disk (interactive book), Auto Run Demo Disk, Reference Card and COMAL FROM A TO Z book.
  $29.95 plus $2 handling

(2) PROFESSIONAL COMAL Version 2.0

- Full 64K Commodore 64 Cartridge
  Twice as Powerful, Twice as Fast
  $99.95 plus $2 handling (no manual or disks)
- Deluxe Cartridge Package includes:
  COMAL HANDBOOK 2nd Edition, Graphics and Sound Book, 2 Demo Disks and the cartridge (sells for over $200 in Europe). This is what everyone is talking about.
  $128.90 plus $3 handling (USA & Canada only)

CAPTAIN COMAL™ Recommends:

The COMAL STARTER KIT is ideal for a home programmer. It has sprite and graphics control (LOGO compatible). A real bargain—$29.95 for 3 full disks and a user manual.

Serious programmers want the Deluxe Cartridge Package. For $128.90 they get the best language on any 8 bit computer (the support materials are essential due to the immense power of Professional COMAL).

ORDER NOW:

Call TOLL-FREE: 1-800-356-5324 ext 1307 VISA or MasterCard ORDERS ONLY. Questions and information must call our Info Line: 608-222-4432. All orders prepaid only—no C.O.D. Send check or money order in US Dollars to:

COMAL USERS GROUP, U.S.A., LIMITED
5501 Groveland Ter., Madison, WI 53716

TRADEMARKS: Commodore 64 of Commodore Electronics Ltd; Captain COMAL of COMAL Users Group, U.S.A., Ltd
‡ estimated

WATCOM

Products for the Commodore 64

Waterloo Structured BASIC

Already widely used on the Commodore PET, the package augments the standard BASIC with:

- *Structured Programming Statements* : programs can be written with proper style.
- *Procedures* : eliminate the use of GOSUB; instead CALL named procedures
- *Additional Commands* : increased ease of use with AUTO, DELETE and RENUMBER commands

Each package contains:

- cartridge containing software
- comprehensive textbook containing both a primer and a reference manual

Price: $99.00; $79.00 for additional packages in same order

WATCOM Pascal

This interpreter supports the full ANSI standard Pascal (with one omission) and features:

- integrated full-screen editor
- interactive debugger
- support for printer, disk and cassette
- graphics library
- *peek* and *poke* functions

Each package contains

- cartridge and disk containing the software
- comprehensive textbook containing both a primer and a reference manual

Price: $149.00; $99.00 for additional packages in same order

Ordering Information

Order forms and/or additional information may be obtained from:

WATCOM Products,
415 Phillip Street,
Waterloo, Ontario
Canada, N2L 3X2

(519) 886-3700
Telex: 06-955458

Additional textbooks are also available. Seminars on Pascal and BASIC are offered regularly.

The Transactor
The Tech/News Journal For Commodore Computers

PAYS

$40

per page for articles

We're also looking for professionally drawn cartoons!

Send all material to:

The Editor
The Transactor
500 Steeles Avenue
Milton, Ontario
L9T 3P7

Volume 5 Editorial Schedule

Issue#	Theme	Copy Due	Printed	Release Date
1	Graphics and Sound	Feb 1	Mar 19	April 1
2	The Transition to Machine Code	Apr 1	May 21	June 1
3	Software Protection & Piracy	Jun 1	Jul 23	August 1
4	Business and Education	Aug 1	Sep 17	October 1
5	Hardware and Peripherals	Oct 1	Nov 19	December 1
6	Programming Aids & Utilities	Dec 1	Jan 19	February 1/85

Volume 6 Editorial Schedule

1	Communications & Networking	Feb 1	Mar 21	April 1/85
2	Languages	Apr 1	May 20	June 1
3	Implementing The Sciences	Jun 1	Jul 18	August 1
4	Hardware & Software Interfacing	Aug 1	Sep 21	October 1
5	Real Life Applications	Oct 1	Nov 19	December 1

Advertisers and Authors should have material submitted no later than the 'Copy Due' date to be included with the respective issue.

THE WAIT IS OVER

The Commodore 64™ COMAL 2.0 Cartridge is being produced for Nov 84 delivery.

- Full 64K ROM Cartridge—30K Free User Memory
- Empty socket for user EPROM (8K, 16K or 32K)
- 'LOGO' compatible turtle graphics (with abbreviations)
- Easy Sprite ANIMATION—Load, Save, Link Shapes
- Interrupt driven Music—Full control of SID
- User definable fonts—Load, Save, Link Fonts
- Three different screen dumps in:
  Hi-Res Graphics, Multi-color Graphics, Text
- Error Handler, External Procedures, Trace Commands
- Protected Input, Batch Command File Capability
- Easily definable Function Keys
  (i.e., F7 will RUN any program from a directory)
- Built in LINK command for Machine Code Routines
- HEX and BINARY accepted—ASCII conversion built in

All prepaid advance orders receive FREE:

- COMAL HANDBOOK, Second Edition (includes Cartridge)
- Introduction to 2.0 Built in Packages Book
  (Graphics, Turtle, Sprites, Sound, Font, . . .)
- TWO different demo disks (1541 format)
- White custom molded case for disks and book

ALL FOR ONLY $99.95
(A $175 value—nearly half price)

Due to high demand orders will be filled on first come basis.
Send check or Money Order in US Dollars plus $3 handling to:

COMAL USERS GROUP, U.S.A., LIMITED
5501 Groveland Ter., Madison, WI 53716
phone: (608) 222-4432

VISA and MC prepaid orders may call toll free: 1-800-356-5324
extension 1307

Commodore 64 is a trademark of Commodore Electronics Ltd

The MIDNITE SOFTWARE GAZETTE

The PAPER

Five years of service to the PET community.

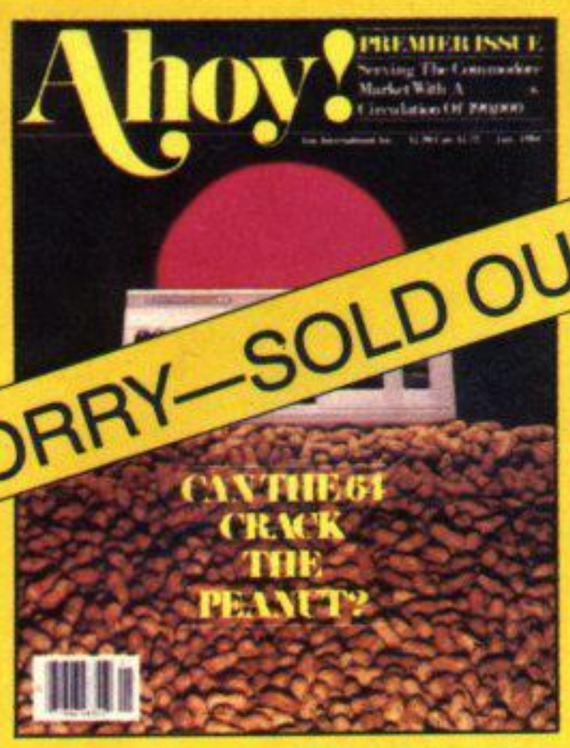
The Independent U.S. magazine for users of Commodore brand computers.

EDITORS: Jim and Ellen Strasma
Sample issue free on request, from:

635 MAPLE □ MT. ZION, IL 62549 USA

Ahoy!

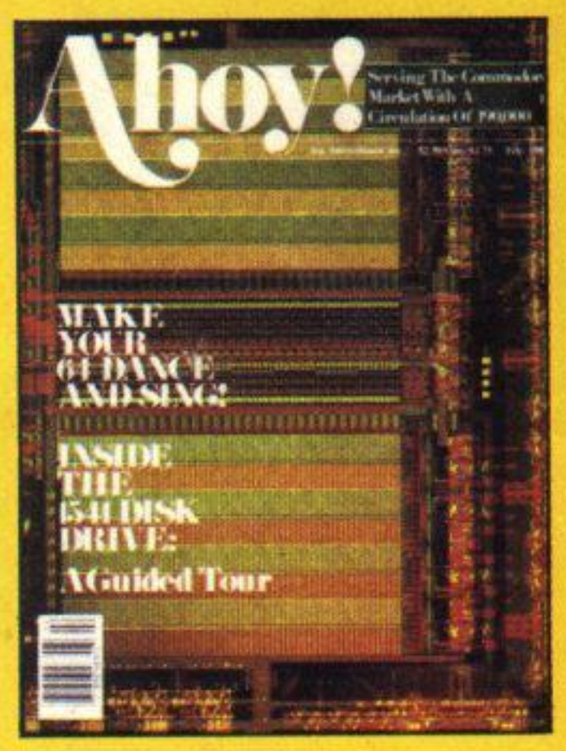
Back Issues



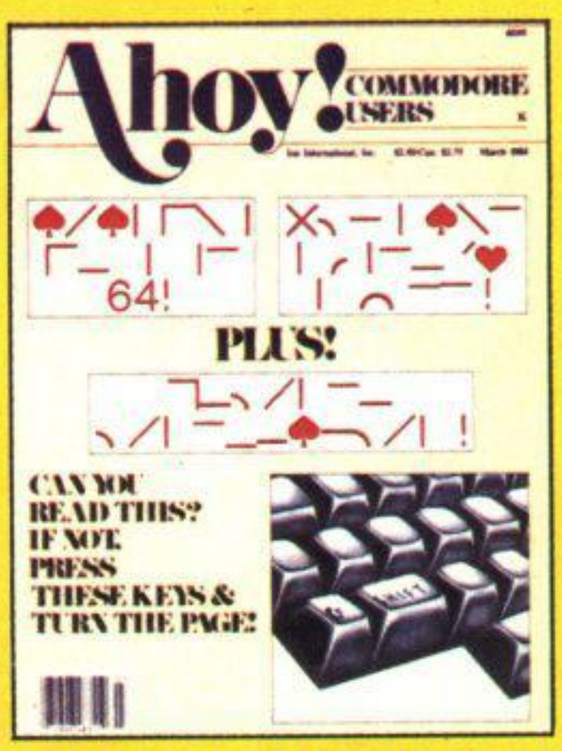
SORRY—SOLD OUT

ISSUE #1—JAN. '84 $4.00
The 64 v. the Peanut! The computer as communications device! Protecto's Bill Badger interviewed! And ready to enter: the Multi Draw 64 graphics system! The Interrupt Music Maker/ Editor! A Peek at Memory! Programming Sequential Files!

Don't punch another key without a complete collection of *Ahoy!* and the programming strategies and product analyses each issue provides. Order while supplies last!

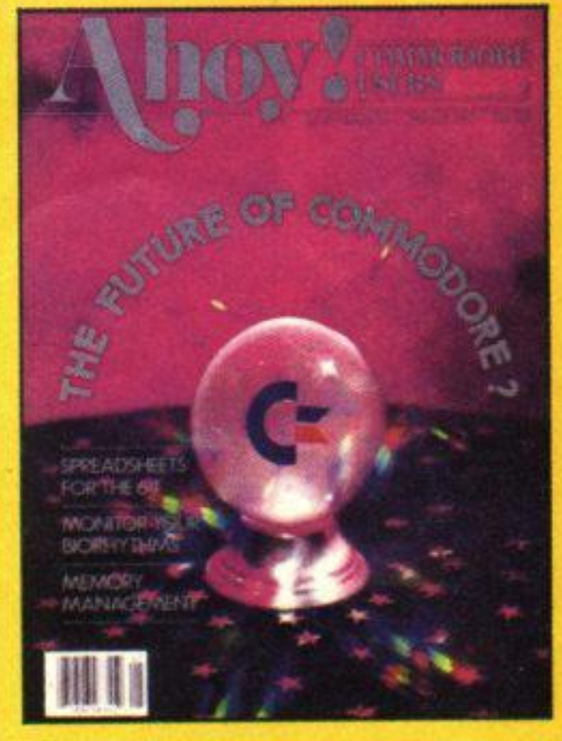


ISSUE #2—FEB. '84 $4.00
Illustrated tour of the 1541 disk drive! Synapse's Ihor Wolosenko interviewed! Users groups! Artificial intelligence! And ready to enter: Music Maker Part II! Night Attack! Programming Relative Files! Screen Manipulation on the Commodore 64!



ISSUE #3—MAR. '84 $4.00
Anatomy of the 64! Printer Interfacing for the 64 and VIC! Educational software: first of a series! Commodares! And ready to enter: Space Lanes! Random Files on the 64! Easy Access Address Book! Dynamic Power for your 64!

ISSUE #4—APR. '84 $4.00
Petspeed and Easy Script tutorials! Printer interfacing and educational software guide continued! Lower case descenders on your 1525! Laserdisc! The Dallas Quest Adventure Game! And ready to enter: Apple Pie! Lunar Lander! Name that Star!



ISSUE #5—MAY '84 $4.00
The Future of Commodore! Inside BASIC program storage! C-64 Spreadsheets! Memory Management on the VIC and 64! Educational Software Guide continues! And ready to enter: Math Master! Air Assault! Biorhythms! VIC 20 Calculator!

Send coupon or facsimile to:
Ahoy! Back Issues, Ion International Inc., 45 West 34th Street—Suite 407, New York, NY 10001

Ahoy!

Please Send Me The Following:

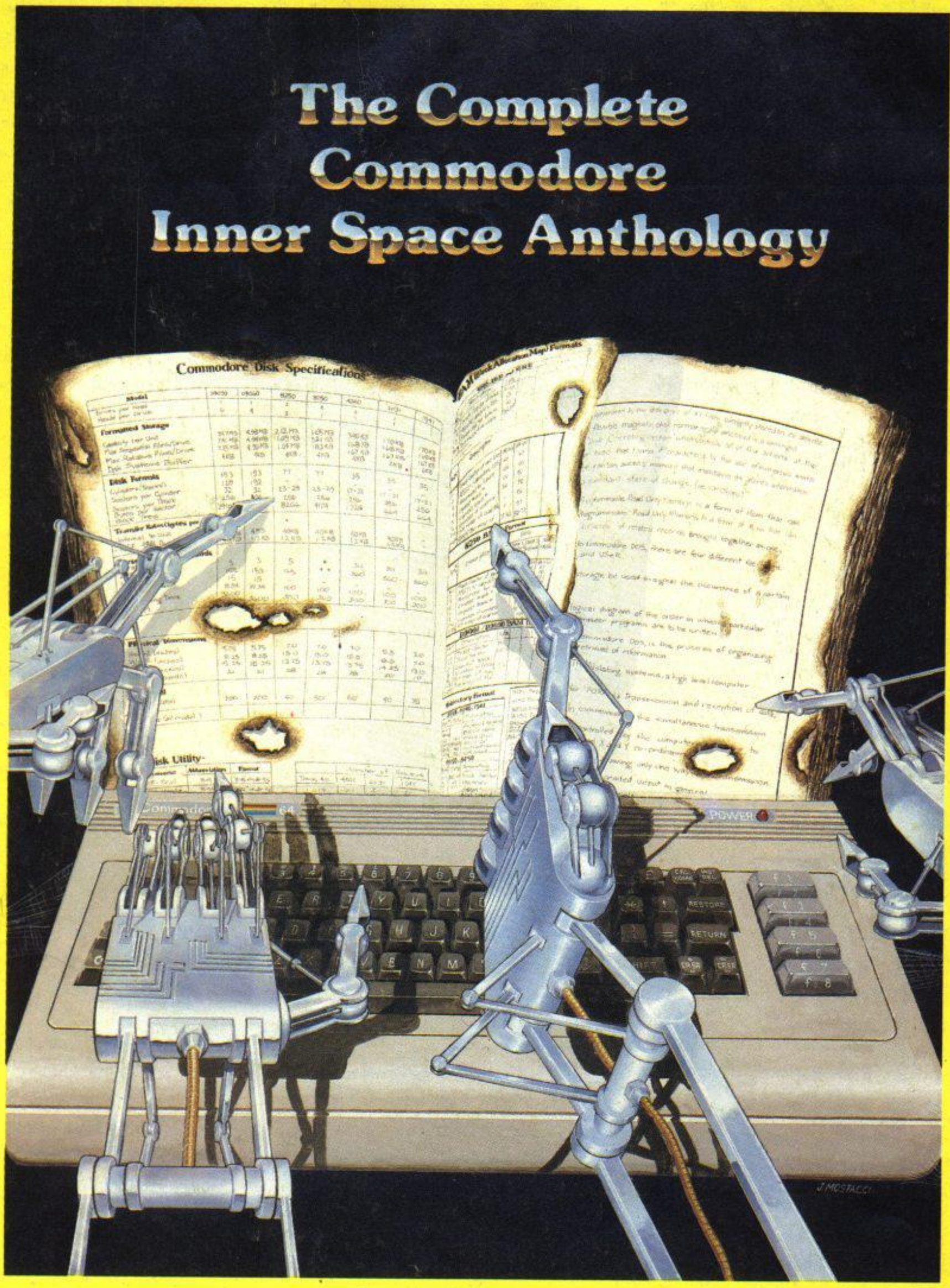
____ Copies of issue number ____
____ Copies of issue number ____
____ Copies of issue number ____

Enclosed Please Find My Check or Money Order for $_____

(Outside the USA please add $1.00 for every copy)

NAME_____
ADDRESS_____
_____
CITY_____
STATE_____
ZIP CODE_____

The Complete Commodore Inner Space Anthology will look like this:



WATCH FOR IT!
January 1985